Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3511

Kompakte und ganzheitliche Visualisierung von Ontologien

David Bold

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Steffen Lohmann, M.Sc.
Beginn am:	14. Juni 2013
Beendet am:	13. Dezember 2013
CR-Nummer:	H.5.2, H.4.0, H.3.4, D.2.2

Kurzfassung

Im Kontext des Semantischen Webs und Linked Data sind Ontologien ein beliebtes Konzept der Wissensrepräsentation für semantische Anreicherung und Strukturierung von Daten. Bisher existieren nur Visualisierungskonzepte zur Darstellung von Teilaspekten von Ontologien. Ein Konzept zur Visualisierung aller Aspekte einer Ontologie fehlt bisher. Im Rahmen dieser Arbeit werden sechs bestehende Visualisierungskonzepte auf ihre jeweiligen Stärken und Schwächen analysiert und darauf aufbauend ein kompaktes und ganzheitliches Visualisierungskonzept vorgestellt und weiterführend optimiert. Dieses Konzept versucht die analysierten Schwächen bestehender Konzepte zu vermeiden und realisiert das entwickelte und optimierte Konzept in Form eines Prototypen. Zur Umsetzung der prototypischen Realisierung werden acht verschiedene Grafikframeworks auf ihre Eignung für die Entwicklung des Prototypen untersucht und anhand in dieser Arbeit aufgestellten Kriterien eines ausgewählt und verwendet. Die Architektur des realisierten Prototypen ist dokumentiert und wurde durch eine Benutzerstudie im Rahmen dieser Arbeit evaluiert. Dabei wurden Stärken und Schwächen des Visualisierungskonzepts herausgearbeitet und mit alternativen Konzepten verglichen.

Abkürzungsverzeichnis

ARPA Advanced Research Projects Agency
RDF Resource Description Framework
RDFS Resource Description Framework Schema
OWL Web Ontology Language
OWL2 Web Ontology Language 2.0
W3C World Wide Web Consortium
MCF Meta Content Framework
URI Uniform Resource Identifier
HTML Hypertext Markup Language
UML Unified Modeling Language
ERM Entity-Relationship Model
BPMN Business Process Model and Notation
VOWL Visual Notation for OWL Ontologies
JUNG Java Universal Network/Graph Framework
FOAF friend-of-a-friend
MUTO modular-unified-tagging-ontology

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1. Einleitung

Die Entwicklung des Internets begann 1966 mit einem Projekt der Advanced Research Projects Agency (ARPA), einer Behörde des Verteidigungsministeriums der Vereinigten Staaten, mit dem Ziel Computer miteinander zu vernetzen. 1970 wurde begonnen, die verschiedenen, bereits existierenden, Teilnetze zu einem großen Netz zusammenzufassen. Bereits in den 1960er Jahren wurden erste Ideen zur Realisierung einer Hypertextstruktur zur Informationsnavigation beschrieben, die die damals jedoch noch nicht real umgesetzt werden konnten [KV06]. Der Gedanke neben Text-Anweisungen und Format-Anweisungen auch Befehle für Hyperlinks [Wik13] zu integrieren, war jedoch schon geboren [KV06], auch wenn der Entwurf durch Tim Berners-Lee erst 1989 öffentlich als Diskussionspapier vorgestellt werden sollte [BL89].
Auch wenn Vordenker bereits damals die Möglichkeiten vernetzter Netzwerke und der Hypertext Sprache erkannten, ist es fraglich, ob sie vorhersehen konnten, dass im Jahre 2012 bereits 34.3 % gesamten der Weltbevölkerung bzw. 63.2 % der Bevölkerung Europas und 78.6 % der Bevölkerung von Nord-Amerika diese Technologie verwendet. Zwischen 2000 und 2012 betrug das Wachstum der Anwender des World Wide Webs 566 % [Min12]. Diese Zahlen verdeutlichen die Evolution einer Idee zu einem weltweit angewandten Medium.
Unser heutiger Alltag ist ohne das World Wide Web kaum mehr vorstellbar, es lässt uns miteinander Meinungen austauschen und bietet uns eine unvorstellbare Menge an Informationen und Unterhaltung verschiedenster Art. Dienstleistungen rund ums World Wide Web sind längst zu einem bedeutenden wirtschaftlichen Faktor geworden, das Netz steuert in Deutschland beispielsweise bereits 21 % des Wachstums des Bruttosozialprodukt bei [Sch11]. Das Datenvolumen des World Wide Web wächst stetig weiter, im Jahre 2011 umfasste es bereits 1,8 Trillionen Gigabytes in fünf hundert Quadrillion „Dateien“, für das Jahr 2015 werden 8 Trillionen Gigabytes prognostiziert [GR11]. 90 % dieser Daten sind jedoch unstrukturiert [GR11], sodass Werkzeuge benötigt werden, um diesen Datenberg zu durchsuchen. Mit dem Semantischen Web wurde von Tim Berners-Lee eine Möglichkeit entworfen, Daten nicht nur miteinander zu vernetzen, sondern Informationen samt ihrer Bedeutung miteinander zu verknüpfen [BL98].
Um Daten zu strukturieren und semantisch anzureichern stellen Ontologien ein populäres Konzept der Wissensrepräsentation dar. Ontologien bilden dabei ein Netz von Hierarchien ab, dabei können Informationen durch logische Beziehungen miteinander verknüpft sein. Aufgrund der gestiegenen Verbreitung der Nutzung von Ontologien in der Wissensrepräsentation stieg auch der Wunsch, Ontologien visuell darzustellen. Dabei muss jedoch beachtet werden, dass kein einheitliches Visualisierungskonzept für Ontologien existiert. Vorhanden ist hingegen eine Vielzahl unterschiedlicher Ansätze, die häufig nur einen Teilaspekt grafisch abbilden können.

1.1. Zielsetzung

Im Rahmen dieser Arbeit soll eine kompakte und ganzheitliche Visualisierung für Ontologien entwickelt und prototypisch umgesetzt werden. Die Visualisierung soll dabei die Konzepte und Relationen der Ontologien verständlich darstellen und einen Eindruck der enthaltenen Daten der Instanzen vermitteln. Die meisten, der in OWL spezifizierten Ontologie-Konstrukte sollen auf kompakte und verständliche Weise dargestellt werden, während Details interaktiv exploriert werden können. Das Visualisierungskonzept soll, mit realen Daten aus einer Nutzerstudie, evaluiert werden. Die Ergebnisse der Studie werden im Anschluss ausgewertet und diskutiert.

1.2. Gliederung

In diesem Abschnitt wird ein Überblick über die Struktur dieser Arbeit vermittelt.

Kapitel 2 – Grundlagen: Dieses Kapitel erläutert die Themenfelder, die dieser Arbeit zugrunde liegen. Hierzu zählen vor allem Ontologien und ihre Darstellung.

Kapitel 3 – Themenverwandte Arbeiten: In diesem Kapitel werden verschiedene, bereits existierende Konzepte zur Visualisierung von Ontologien vorgestellt und ihre Stärken und Schwächen erläutert.

Kapitel 4 – Konzept: In diesem Kapitel wird das verwendete und optimierte Konzept näher erläutert, welches die Vorlage der prototypischen Umsetzung bildet.

Kapitel 5 – Frameworks für Graphen: In diesem Kapitel werden verschiedene Frameworks für Graphen, speziell für Knoten-Kanten-Diagramme vorgestellt und auf ihre Eignung, die spätere prototypische Umsetzung zu unterstützen, untersucht.

Kapitel 6 – Implementierung: In diesem Kapitel wird die Struktur und Architektur der prototypischen Umsetzung beschrieben und das zeitliche Vorgehen während der Implementierung erläutert.

Kapitel 7 – Evaluation: In diesem Kapitel wird das optimierte Konzept und dessen prototypische Umsetzung im Rahmen einer Expertenstudie evaluiert.

Kapitel 8 – Zusammenfassung & Ausblick: Dieses abschließende Kapitel gibt einen zusammenfassenden Überblick über die Arbeit und bietet einen Ausblick auf weitere Aspekte, die in fortführenden Arbeiten behandelt werden könnten.

2. Grundlagen

Dieses Kapitel enthält die Grundlagen, die dem Verständnis dieser Arbeit dienen. Hierzu gehört die Einführung und Vorstellung von Ontologien sowie deren Anwendung und Visualisierung.

2.1. Ontologie

Der Begriff der Ontologie entstammt der Philosophie. Er beschreibt sowohl die Möglichkeiten als auch die Bedingungen des Seienden und setzt sich dabei sowohl mit den Fähigkeiten als auch den Grenzen des menschlichen Wahrnehmens und Erkennens auseinander.
In vielen Bereichen der modernen Welt muss Wissen mit anderen geteilt, Erkanntes und Erdachtes repräsentiert, sowie Sachverhalte, Regeln und Fakten modelliert werden. Menschen verwenden beim Erlernen von Fachwissen verschiedene Hilfsmittel wie Lehrbücher, Regelwerke, Lexika oder Schlagwortregister. In der Regel können Menschen aus unstrukturierten Texten Zusammenhänge und die verwendeten Begriffe erkennen.
Sollen Maschinen Entscheidungen auf Basis ihres gespeicherten Wissens treffen, so wird eine Repräsentation der zugrunde liegenden Begriffe und deren Zusammenhänge benötigt. Hilfsmittel hierfür können Ontologien sein. Ontologien ermöglichen die maschinelle Wiederverwendung von Wissen und die Erstellung automatischer Schlussfolgerungen [Hes02].
Tom Gruber definierte eine Ontologie 1992 als formale Spezifikation einer Konzeptualisierung.

> „An ontology is a specification of a conceptualization“ [G+93]

Ontologien beschreiben einen Wissensbereich mithilfe standardisierter Terminologien und definieren die Beziehungen und Ableitungsregeln zwischen den zuvor definierten Begriffen. Klassen, Individuen, Relationen, Funktionen und Axiome sind ihre Hilfsmittel. [G+93].
Innerhalb einer Ontologie werden Axiome als Aussagen bezeichnet, die immer erfüllt und damit wahr sein müssen. Mithilfe von Axiomen lassen sich beispielsweise Vererbungsregeln zwischen Klassen und Relationen definieren. Enthält eine Ontologie Axiome, kann sie als schwergewichtig eingeteilt werden, andernfalls als leichtgewichtig.
Relationen sind gerichtet und verweisen auf Klassen oder Individuen. Sätze entstehen durch das Verknüpfen von Informationen. Informationen setzen sich immer aus einem Tripel bestehend aus Subjekt, Prädikat und Objekt zusammen [Hit07]. Abbildung 2.1 verdeutlicht dies anhand eines Beispiels.

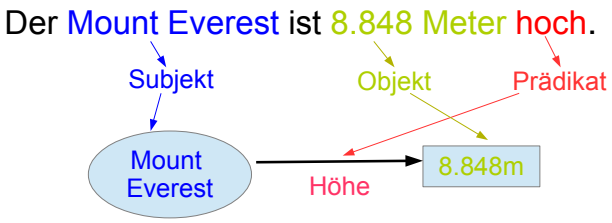


Abbildung 2.1.: Beispiel eines Tripels aus Subjekt, Prädikat und Objekt und dessen Abbildung als Relation.

Veranschaulichung anhand eines Beispiels

Als Beispiel sei eine Ontologie gewählt, die die Existenz von vier Personen, von denen je zwei weiblich und männlich sind, modelliert. Diese Ontologie kann, wie in Quelltext 2.1 gezeigt, textuell dargestellt werden. In diesem Beispiel wurden die Klassen „Person“, „Frau“ und „Mann“ definiert. „Frau“ und „Mann“ wurden als Unterklasse der Klasse „Person“ definiert. Beide enthalten jeweils zwei Individuen. Diese Ontologie hat als Axiom, dass die Teilmenge von männlich und weiblich nur die leere Menge enthält, sie sind disjunkt.
Die in Quelltext 2.1 gezeigte Ontologie wurde mittels eines Werkzeugs erstellt, das in Abschnitt 2.6 näher beschrieben wird. Das in Quelltext 2.1 verwendete Dateiformat wird in den Abschnitten 2.2, 2.3 und 2.4 näher beschrieben.

Quelltext 2.1: Textuelle Darstellung der Ontologie.

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
 <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
 <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
 <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
 <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#"
 xml:base="http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:owl="http://www.w3.org/2002/07/owl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

 <owl:Ontology rdf:about=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146"/>

 <owl:Class rdf:about=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Person"/>

 <owl:Class rdf:about=
```

```
  "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Mann">
 <rdfs:subClassOf rdf:resource=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Person"/>
</owl:Class>

 <owl:Class rdf:about=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Frau">
  <rdfs:subClassOf rdf:resource=
    "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Person"/>
    <owl:disjointWith rdf:resource=
      "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Mann"/>
   </owl:Class>

<owl:NamedIndividual rdf:about=
  "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#P1">
 <rdf:type rdf:resource=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Frau"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about=
  "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#P2">
 <rdf:type rdf:resource=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Frau"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about=
 "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#P3">
  <rdf:type rdf:resource=
    "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Mann"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about=
  "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#P4">
 <rdf:type rdf:resource=
   "http://www.example.org/db/ontologies/2013/11/untitled-ontology-1146#Mann"/>
</owl:NamedIndividual>
</rdf:RDF>
```

Ontologien werden häufig visuell dargestellt. Dies kann beispielsweise mithilfe der Werkzeuge OWLViz (Abbildung 2.2), OntoGraf (Abbildung 2.3) und SOVA (Abbildung 2.4) geschehen. Die verfügbaren Werkzeuge unterscheiden sich im Umfang ihrer Darstellung, so stellt OWLViz beispielsweise keine Individuen dar. OntoGraf visualisiert hingegen die Wurzel aller Klassen, `owl:Thing` nicht. Alle drei hier gezeigten Visualisierungen werden in Kapitel 3 detailliert beschrieben.

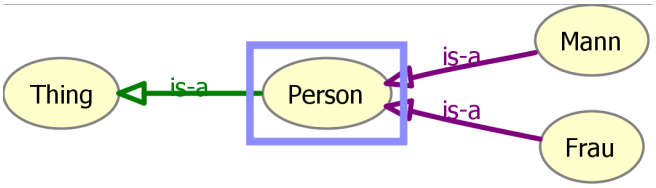


Abbildung 2.2.: Ontologie aus Quelltext 2.1 visualisiert mittels OWLViz.

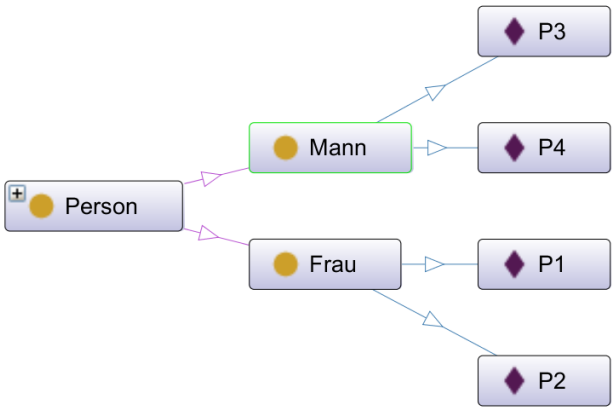


Abbildung 2.3.: Ontologie aus Quelltext 2.1 visualisiert mittels OntoGraf.

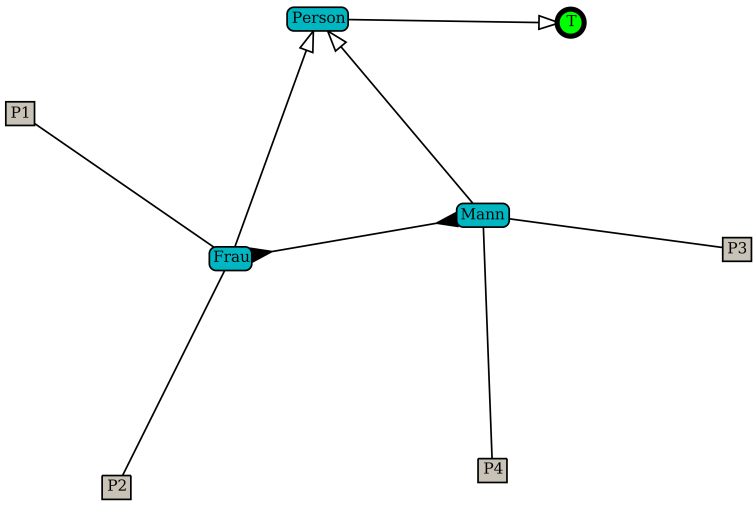


Abbildung 2.4.: Ontologie aus Quelltext 2.1 visualisiert mittels SOVA [PK12].

2.2. RDF

Das Resource Description Framework (RDF) ist ein System, welches vom World Wide Web Consortium (W3C) zur Beschreibung von Metadaten entwickelt wurde. RDF basiert zum Teil auf dem von Ramanathan V. Guha 1995 entworfenen Meta Content Framework (MCF) [Int13] und ist dabei eines von vielen Methoden zur Formalisierung logischer Ausdrücke über beliebige Dinge. Damit ist es, genau wie die darauf aufbauenden Resource Description Framework Schema (RDFS) und Web Ontology Language (OWL), ein Grundbaustein des Semantischen Webs. RDF verwendet aus UML-Klassendiagrammen und Entity-Relationship-Modellen bekannte Methoden zur Modellierung von Konzepten.
Jede Elementaraussage in RDF ist ein Tripel bestehend aus Subjekt, Prädikat und Objekt.

Das Prädikat und das Objekt sind Ressourcen, die das Subjekt näher beschreiben. Ein Objekt kann dabei eine beliebige Ressource sein. Alle Bezeichner in RDF müssen eindeutig sein, daher werden hierfür Uniform Resource Identifier (URI) verwendet [BL94]. Ontologien lassen sich mit RDF formalisieren, sie werden im RDF-Format meist mit dem Mime-Type XML abgespeichert.

2.3. RDF-Schema

RDFS ist eine Abkürzung für das Resource Description Framework Schema, welches ein von der W3C entworfenes Hilfsmittel zur Formalisierung von Ontologien ist. Es bietet ein Vokabular für die Modellierung von Ontologien. RDFS repräsentiert die vorkommenden Ressourcen durch deren Eigenschaften und Relationen. RDFS greift dabei auf die Idee des mengentheoretischen Klassenmodells zurück, bei der Klassen und Eigenschaften getrennt voneinander modelliert werden. RDFS definiert hierzu Klassen und Eigenschaften. Klassen werden in RDFS durch die Attribute `rdfs:Resource`, `rdfs:Class`, `rdfs:Literal`, `rdfs:Datatype`, `rdf:XMLLiteral` und `rdf:Property` näher beschrieben, während Eigenschaften durch `rdfs:range`, `rdfs:domain`, `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:label` und `rdfs:comment` näher definiert werden. Des Weiteren enthält RDFS weitere Begriffe wie `Container Classes` und `RDF Collections` [VVVM04]. RDFS stellt eine abwärts kompatible Schemaerweiterung für RDF dar.

rdfs:Resource: alles innerhalb von RDFS ist eine, durch eine URI eindeutig bestimmte, Ressource.

rdfs:Class: eine Klasse fasst eine Menge gleichartiger Attribute zusammen, sie können Instanzen anderer Klassen sein.

rdfs:Literal ist innerhalb von RDFS ein atomarer Wert, beispielsweise ein String oder ein Zahlenwert.

rdfs:subClassOf sind vergleichbar mit vererbten Klassen. `SubClassOf` ermöglicht Vererbungshierarchien, sie sind transitive Eigenschaften. Falls X eine Unterklasse von Y ist, erbt X alle Eigenschaften von Y. Im Beispiel von Abbildung 2.1 wurde die Klasse „weiblich“ als Unterklasse von „Person“ definiert; dies wurde beispielsweise in Abbildung 2.2 visualisiert.

rdfs:subPropertyOf ermöglichen die Vererbung von Eigenschaften, die äquivalent zu `rdfs:subClassOf` sind.

rdfs:label ermöglicht die Festlegung eines Labels, das eine für Menschen besser lesbaren Version der eindeutigen URI entspricht.

rdfs:comment ermöglicht die Angabe einer für Menschen besser verständlichen Beschreibung einer Ressource.

rdfs:domain legt das Subjekt einer Relation (und deren Typ) fest, Abbildung 2.5 verdeutlicht dies anhand eines Beispiels. Falls eine Eigenschaft eine `rdfs:domain` Angabe enthält, so wird sowohl der Definitionsbereich der Eigenschaft eingeschränkt als auch eine Aussage über die Klasse, die den Definitionsbereich darstellt, getroffen.

rdfs:range legt das Objekt einer Relation (und deren Typ) fest, Abbildung 2.5 verdeutlicht dies anhand eines Beispiels. Falls eine Eigenschaft eine `rdfs:range` Angabe enthält, so wird sowohl Definitionsbereich der Eigenschaft eingeschränkt als auch eine Aussage über die Klasse, die den Definitionsbereich darstellt, getroffen.

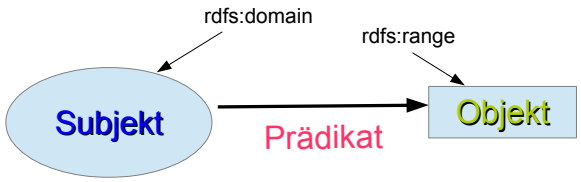


Abbildung 2.5.: `rdfs:domain` und `rdfs:range` an einem Beispiel.

2.4. OWL

Die Web Ontology Language ist eine Spezifikation des W3C, sie stellt eine formale Beschreibungssprache dar mit deren Hilfe Ontologien erstellt, publiziert und verteilt werden können. Zusammen mit RDF und RDFS ist OWL ein Bestandteil der Semantic-Web-Initiative von Tim Berners-Lee. Tim Berners-Lee ist ein Begründer des W3C, derzeitiger Direktor[1] des W3C, Erfinder der Hypertext Markup Language (HTML) und lehrt am Massachusetts Institute of Technology. OWL erweitert die Ausdrucksmöglichkeit von RDFS und RDF. Dabei bleibt OWL weiterhin kompatibel zu RDFS und RDF, denn RDF bzw. RDFS wird durch OWL nur mit zusätzlichen Elementen erweitert. Jedes RDF- und jedes RDFS-Dokument wird durch die OWL Erweiterung weiterhin als valide angesehen. Diese Erweiterung durch OWL ermöglicht beispielsweise die Interferenzbildung und die Formalisierung von Ausdrücken der Prädikatenlogik. OWL kann im Gegensatz zu RDF eine symmetrische, transitive, funktionale und inverse Property beschreiben. Auch ist die Angabe einer Kardinalität in OWL möglich. OWL kann Inferenzen bilden und unterstützt Mengenoperatoren. Da unterschiedliche Anwendungsgebiete verschiedene Zielsetzungen haben, wurden vom W3C mehrere Versionen von OWL definiert [Hit07]:

OWL Lite: ist eine Teilmenge von OWL DL und ist vor allem für einfache Taxonomien gedacht. Taxonomien in OWL Lite sind entscheidbar, es kann stets entschieden werden, ob eine Aussage aus einer Ontologie geschlossen werden kann oder nicht. Viele Sprachelemente aus RDFS sind entweder verboten oder nur stark eingeschränkt zugelassen, beispielsweise sind bei Zahlenrestriktionen nur die beiden Zahlenwerte 1,0 erlaubt.

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

OWL DL: ist eine Teilmenge von OWL Full, sie enthält alle Bestandteile von OWL Lite. Sie wird von den meisten Werkzeugen unterstützt. Innerhalb von OWL DL kann entschieden werden, ob eine Aussage aus der Ontologie geschlossen werden kann. Es wird explizit zwischen Klassen, Individuen, abstrakten und konkreten Rollen und Datentypen unterschieden, zugelassen sind nicht alle Sprachelemente aus RDFS.

OWL Full: enthält die vollständige Ausdrucksstärke von OWL und beinhaltet daher ganz RDFS. Im Gegensatz zu OWL DL und OWL Lite ist OWL Full unentscheidbar, das heißt, es kann nicht entschieden werden, ob eine Aussage aus einer Ontologie geschlossen werden kann. Hauptsächlich basiert die Unentscheidbarkeit auf der Möglichkeit die Typen Individuen, Klassen und Rollen nicht zu trennen. Nach [Hit07] wird OWL Full von Werkzeugen derzeit nur bedingt unterstützt.

OWL erweitert sowohl RDF als auch RDFS und ermöglicht beispielsweise die Formulierung abzubilden, dass Individuen (Instanzen einer Klasse) unterschiedlich zueinander sind. Hierfür bietet OWL verschiedene Möglichkeiten, siehe hierzu Quelltext 2.2 und Quelltext 2.3. Die Aussage „Oper1, Oper2 und Oper3 sind unterschiedliche Opern“ wird in beiden Beispielen formalisiert. In Quelltext 2.2 enthält „Oper2“ die Angabe, dass sie ungleich „Oper1“ ist. „Oper3“ beinhaltet die Divergenz zu „Oper1“ und „Oper2“. Damit kann „Oper1“ ebenfalls nicht identisch zu „Oper2“ oder „Oper3“ sein. Die Spezifikation dieser Verschiedenartigkeit erfolgt durch das OWL-Element `<owl:differentFrom rdf:resource=URI/>` innerhalb der jeweiligen Oper. In Quelltext 2.3 wird dieselbe Aussage formalisiert. Hierbei erfolgt dies jedoch nicht als Angabe innerhalb der jeweiligen Oper, sondern als eigenständige Aussage. Durch `<owl:AllDifferent>` wird ausgedrückt, dass folgende Auflistung verschiedenartig ist. `<owl:distinctMembers rdf:parseType="Collection»` definiert die zuvor erwähnte Auflistung, die wiederum die Elemente `Oper1`, `Oper2` und `Oper3` beinhaltet.

Quelltext 2.2 Eine Möglichkeit zur Formalisierung der Aussage: drei verschiedene Opern sind verschieden.

```
<Opera rdf:ID="Oper1"/>

<Opera rdf:ID="Oper2">
  <owl:differentFrom rdf:resource="#Oper1"/>
</Opera>

<Opera rdf:ID="Oper3">
  <owl:differentFrom rdf:resource="#Oper1"/>
  <owl:differentFrom rdf:resource="#Oper2"/>
</Opera>
```

Quelltext 2.3 Eine andere Möglichkeit zur Formalisierung der Aussage: drei verschiedene Opern sind verschieden.

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Opera rdf:about="#Oper1"/>
    <Opera rdf:about="#Oper2"/>
    <Opera rdf:about="#Oper3"/>
  </owl:distinctMembers>
</owl:AllDifferent>
```

Eine Klasse wird in OWL gemäß Quelltext 2.4 beschrieben. Sie enthält eine eindeutige URI, ein für Menschen besser lesbares Label als Alternative zur URI und ein Kommentar als Beschreibung. Der Kommentar spielt nur für Menschen eine Rolle, die Ontologie wird weder vom Label noch vom Kommentar verändert. Die URI wird durch `<owl:Class rdf:about=URL`, das Label durch `<rdfs:label xml:lang=ISO-CODE>` spezifiziert. Die Angabe `<rdfs:subClassOf`[2] sagt aus, dass diese Klasse eine Unterklasse von „Item“ ist und daher dieselben Eigenschaften, wie „Item“ enthält.

Quelltext 2.4 Eine Klasse aus OWL, entnommen aus der MUTO-Ontologie [LDA11].

```
 <owl:Class rdf:about="http://purl.org/muto/core#Tagging">
    <rdfs:label xml:lang="en">Tagging</rdfs:label>
    <rdfs:comment xml:lang="en">A tagging links a resource to a user account and one or more
        tags.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://rdfs.org/sioc/ns#Item"/>
        <rdfs:isDefinedBy rdf:resource="http://purl.org/muto/core#"/>
  </owl:Class>
```

Quelltext 2.5 beschreibt eine Eigenschaft nach OWL. Sie enthält eine eindeutige URI, ein für Menschen besser lesbares Label als Alternative zur URI und einen Kommentar als Beschreibung. `rdfs:range` beschreibt das Objekt dieser Relation während `rdfs:domain` das Subjekt dieser Relation beschreibt. Falls die Angabe einer `rdfs:range` bzw. einer `rdfs:domain` fehlt, so wird diese Eigenschaft nicht eingeschränkt. In diesem Fall bezieht sich die Relation auf `OWL:Thing` und damit auf alle Subjekte innerhalb einer Ontologie.

Quelltext 2.5 Eine Eigenschaft, der MUTO-Ontologie entnommen [LDA11].

```
<owl:ObjectProperty rdf:about="http://rdfs.org/sioc/ns#account_of">
  <rdfs:label xml:lang="en">account of</rdfs:label>
  <rdfs:comment xml:lang="en">Refers to the foaf:Agent or foaf:Person who owns this
      sioc:UserAccount.</rdfs:comment>
  <rdfs:domain rdf:resource="http://rdfs.org/sioc/ns#UserAccount"/>
  <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Agent"/>
  <rdfs:isDefinedBy rdf:resource="http://rdfs.org/sioc/ns#"/>
  <owl:inverseOf rdf:resource="http://xmlns.com/foaf/0.1/account"/>
</owl:ObjectProperty>
```

[2] http://www.w3.org/TR/owl-ref/#subClassOf-def

Quelltext 2.6 beschreibt ebenfalls eine Eigenschaft. Diese Eigenschaft bezieht sich im Gegensatz zu Quelltext 2.5 nicht auf eine andere Klasse, sondern auf ein Literal und damit auf einen atomaren Wert. Ansonsten verhält sich Quelltext 2.6 analog zu Quelltext 2.5.

Quelltext 2.6 Eine Eigenschaft, der MUTO-Ontologie entnommen [LDA11].

```
<owl:DatatypeProperty rdf:about="http://rdfs.org/sioc/ns#content">
  <rdfs:label xml:lang="en">content</rdfs:label>
  <rdfs:comment xml:lang="en">The content of the Item in plain text format.</rdfs:comment>
  <rdfs:domain rdf:resource="http://rdfs.org/sioc/ns#Item"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  <rdfs:isDefinedBy rdf:resource="http://rdfs.org/sioc/ns#"/>
</owl:DatatypeProperty>
```

Falls eine Klasse als Unterklasse einer anderen Klasse modelliert werden soll, so kann dies auf unterschiedliche Art und Weise realisiert werden. Die Klasse kann, wie in Quelltext 2.4 gezeigt, direkt innerhalb der Klassendefinition als Unterklasse einer anderen Klasse definiert werden. Die Unterklassendefinition kann, wie in Quelltext 2.7 gezeigt, auch eigenständig erfolgen.

Quelltext 2.7 Eigenständige SubClassOf Definition.

```
  <SubClassOf>
     <Class IRI="#SubClass"/>
     <Class IRI="#ParentClass"/>
  </SubClassOf>
```

Abbildung 2.6 stellt eine mögliche Visualisierung der MUTO-Ontologie [LDA11] dar. Die in Quelltext 2.4 beschriebene Klasse ist in dieser Visualisierung hervorgehoben. Die Eigenschaften dieser Klasse werden hier als Tooltip wiedergegeben. Die `Object Property` aus Quelltext 2.5 und die `Datatype Property` aus Quelltext 2.6 werden in dieser Visualisierung nur in Form von Pfeilen dargestellt.

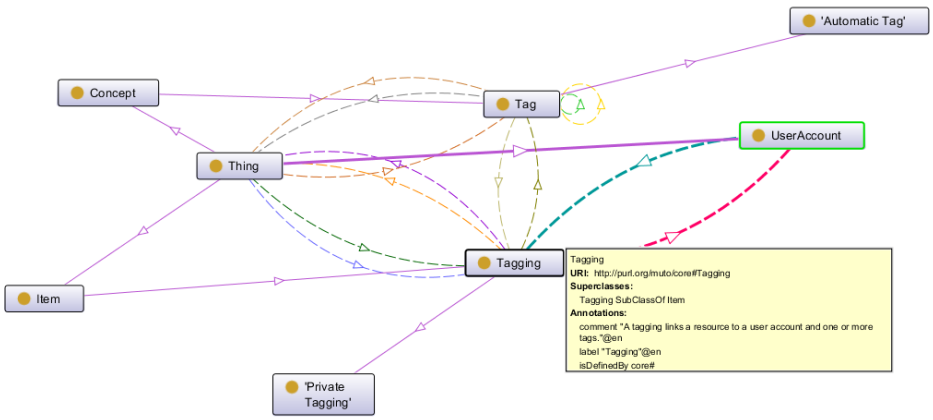


Abbildung 2.6.: Eine Visualisierung der MUTO-Ontologie [LDA11] erstellt durch OntoGraf.

Einen vollständigen Überblick über die Ausdrucksstärke und Mächtigkeit von OWL bietet die W3C Empfehlung zu OWL [BVHH+04]. Dieses Dokument enthält auch eine ausführliche Beschreibung der einzelnen Parameter. Eine neuere Version der OWL wurde in Form der Web Ontology Language 2.0 (OWL2) bereits veröffentlicht [MGH+09]. OWL2 erweitert die Ausdrucksmöglichkeiten von OWL um weitere Features, die von der W3C[3] aufgelistet werden, und enthält ebenfalls verschiedene Versionen für unterschiedliche Anwendungszwecke: OWL 2 EL, OWL 2 QL, und OWL 2 RL. Die Gruppeneinteilung bei OWL2 erfolgt anhand der Laufzeit, OWL 2 EL steht zum Beispiel für Polynomialzeitalgorithmen und eignet sich beispielsweise für die Bildung von Schlussfolgerungen innerhalb von Polynomialzeit.

2.5. OWL-API

Die OWL-API ist eine 2003 erschienene JAVA Referenzimplementierung, mit deren Hilfe OWL-Ontologien erstellt, geändert und serialisiert werden können. Die aktuellste Fassung der OWL-API konzentriert sich auf die Version 2 von OWL, sie wird unter anderem von Protégé 4 verwendet und unterstützt Protégé beim serialisieren [HB11]. Unterstützt werden die folgenden Formate:

- RDF/XML
- OWL/XML
- Terse RDF Triple Language
- OWL Functional Syntax
- Manchester OWL Syntax
- KRSS Syntax
- ODA Flat File

2.6. Protégé

Protégé [Ins13a] ist ein Editor zur Modellierung von Ontologien. Seine Entwicklung begann am Institut für Medizinische Informatik der Universität Stanford. Ziel der Entwicklung war es, einen Editor zum Erstellen von Ontologien für medizinische Forschungszwecke zu schaffen. Entwickelt wird Protégé in der Programmiersprache JAVA und seine Veröffentlichung geschieht unter der Mozilla Public License [Moz13]. Mit Protégé kann eine Wissensdatenbank angelegt, diese mit Informationen befüllt und Wissen daraus abgefragt werden. Aktuell werden drei verschiedene Versionen von Protégé angeboten [Ins13b]:

[3] http://www.w3.org/TR/2009/REC-owl2-quick-reference-20091027/#New_Features_in_OWL_2

- Protégé Desktop 3
- Protégé Desktop 4
- WebProtégé

Protégé Desktop 3 bzw. Protégé Desktop 4 werden innerhalb der Protégé Dokumentation auch als P3 bzw. P4 bezeichnet.

Protégé 3

In Protégé 3 können Ontologien auf zwei unterschiedliche Arten modelliert werden:

Protégé-Frames: Informationen werden über eine bestimmte Domäne in einer hierarchischen Struktur modelliert. Protégé arbeitet dabei mit einem frame-basierten Ansatz, der die hierarchischen Strukturen durch Konzepte, Attribute und Instanzen darstellt. Die Schnittstelle von Protégé zum Zugriff auf Ontologien ist dabei mit dem Protokoll Open Knowledge Base Connectivity kompatibel [Ins13d].

Protégé-OWL: Mittels Protégé-OWL können Ontologien für das Semantische Web modelliert werden. Zum Einsatz kommen dabei häufig die W3C Spezifikation RDFS[VVVM04] und OWL[BVHH+04]. Eine OWL-Ontologie kann dabei Klassen, Eigenschaften und Instanzen enthalten. Im Gegensatz zu Protégé-Frames kann Protégé-OWL auch logische Mechanismen verwenden. Dabei kann das implizit enthaltene Wissen durch Schlussfolgerungen erweitern werden [Ins13e]. Protégé-OWL unterstützt nur OWL 1.0 [Ins13b].

Protégé 4

Protégé in Version 4 unterstützt nur noch Protégé-OWL, Protégé-Frames ist nicht mehr enthalten. Protégé 4 unterstützt OWL 2.0 [Ins13b]. Des Weiteren sind in Version 4 einige Verbesserungen enthalten, beispielsweise werden SPARQL-Abfragen ebenso wie die Javacode-Generierung aus Ontologien[4] unterstützt. Das Einlesen einer OWL-Ontologie, welche mit Protégé-OWL aus Protégé 4 abgespeichert wurde, ist mit Protégé-OWL aus Protégé 3 nicht immer fehlerfrei möglich.

[4] http://protegewiki.stanford.edu/wiki/P4_2_Release_Announcement

WebProtégé

WebProtégé ist eine spezielle Version von Protégé , die direkt im Browser ausgeführt werden kann. Serverseitig läuft WebProtégé in einen Servlet Container und benötigt daher einen Server für Java-Web-Anwendungen, beispielsweise Apache Tomcat. Des Weiteren benötigt WebProtégé die NoSQL Datenbank mongoDB[5]. Neben dem Download von WebProtégé im Dateiformat „WAR“ kann WebProtégé auch online ausprobiert werden[6]. WebProtégé ist vom Funktionsumfang derzeit noch stark eingeschränkt, die Version 2 von OWL wird aber bereits unterstützt.

Protégé Plug-ins

Protégé stellt ein Framework dar, welches durch Plug-ins erweitert werden kann. Aufgrund der Änderungen zwischen Protégé 3 und Protégé 4 sind die meisten Plug-ins nur in einer der beiden Protégé Versionen lauffähig. Sie können über die jeweilige Protégé Webseite bezogen [Ins13a] [Ins13a] werden. Plug-ins für Protégé 3 können nach den Kategorien

- Protégé Client-Server [Pro13a]
- Protégé-Frames [Pro13b]
- Protégé-OWL [Pro13c]

unterschieden werden. Plug-ins der Kategorie Protégé-Frames verwenden die Protégé API während Plug-ins der Kategorie Protégé-OWL die Protégé-OWL API, verwenden. Plug-ins für Protégé 4 müssen durch den Wegfall von Protégé-Frames nicht mehr in verschiedene Kategorien eingeteilt werden. Des Weiteren verwenden sie eine neuere Version der Protégé-OWL API [Ins13c], daher sind Protégé 4 Plug-ins inkompatible zu Protégé 3 Plug-ins.

2.7. Graphen

Ein Graph ist eine aus Knoten und Kanten bestehende Struktur. Kanten eines Graphen können Gewichte enthalten, sie können gerichtet oder ungerichtet sein. Knoten repräsentieren meist Werte. Formal lässt sich ein Graph wie folgt definieren [Lei13, Seite 11]:

Definition 1 *Ein Graph ist ein Paar $G = (V, E)$ disjunkter Mengen mit $E \subseteq [V]^2$; die Elemente von E sind also 2-elementige Teilmengen von V. Die Elemente von V nennt man die Ecken (oder Knoten) des Graphen G, die Elemente E seine Kanten.*

[5] http://www.mongodb.org/
[6] http://webprotege.stanford.edu

Menschen können visuelle Informationen meist besser verarbeiten, daher werden Graphen oft visualisiert, hierfür eignen sich vor allem Knoten-Kanten-Diagramme. Die Visualisierung von Graphen findet unter anderem in der Informationsvisualisierung Anwendung. Knoten-Kanten-Diagramme sollen helfen, Prozessabläufe und Beziehungen zu veranschaulichen, den Zugang zu Massendaten durch Strukturierung zu erleichtern und die Mustererkennung der Menschen unterstützen damit Relationen und Strukturen erkannt und in Kontext zu anderen Informationen gesetzt werden können.

> „Die Visualisierung entspricht der Neigung der menschlichen Spezies und unserer Kultur, visuelle Informationsprozesse und Präsentationsformen zu bevorzugen." [Luc13]

In der Visualisierung von Ontologien werden häufig Knoten-Kanten-Diagramme eingesetzt. Beispiele zur Visualisierung einer OWL Ontologie wurden bereits in Abschnitt 2.1 vorgestellt. Im Allgemeinen ist die textuelle Darstellung kompakter und maschinell gut lesbar, während sie für Menschen weder besonders intuitiv noch besonders explorierbar ist, da grafische Visualisierungen für Menschen meist leichter zu verstehen sind.
Im weiteren Verlauf der Arbeit ist mit der Bezeichnung des Graphen die visuelle Repräsentation eines Graphen als Knoten-Kanten-Diagramms gemeint.

2.7.1. Anforderungen an die Darstellung von Graphen

Ein Graph soll Menschen bei der Erfassung der dargestellten Informationen unterstützen. Menschen finden Graphen mit bestimmten Kriterien ästhetische ansprechend [Sim96] [CT98]. Diese können jedoch nicht immer eingehalten werden.

- Schnitte zwischen Kanten und Knoten sollte vermieden werden.
- Schnitte zweier Kanten sollten vermieden werden.
- Sofern ein sinnvoller Mindestabstand zwischen Knoten eingehalten wird, kann die vom Graphen benötigte Fläche minimiert werden. Alternativ zum Mindestabstand können die Knoten auch auf einem festgelegten Gitter angeordnet werden.
- Sofern ein sinnvoller Mindestabstand zwischen Knoten eingehalten wird, kann die Länge der Kanten minimiert werden. Auch in diesem Fall kann ein festgelegtes Gitter eine Alternative zum Mindestabstand darstellen.
- Die Kanten sollten eine ähnliche Länge aufweisen.
- Enthält der Graph eine Symmetrie, so sollte diese Symmetrie auch visualisiert werden.

Graphen mit wenigen Elementen lassen sich selbst unter Berücksichtigung dieser Anforderungen relativ einfach realisieren. Menschen berücksichtigen viele dieser Kriterien intuitiv, denn sie entsprechen größtenteils denen einer ästhetischen Darstellung. Sollen große Graphen mit vielen Elementen maschinell dargestellt werden, so werden Algorithmen benötigt die viele dieser Anforderungen berücksichtigen. Dies wird im Beispiel Abbildung 2.7 berücksichtigt.

Kanten schneiden sich kaum, es gibt nur Schnittpunkte zwischen verschiedenen Verkehrsmitteln (U-Bahn und S-Bahn), Knoten (Haltestellen) werden nicht von Kanten geschnitten und Knickpunkte von Kanten sind selten vorhanden. Kanten von gleichen Verkehrsmitteln haben größtenteils dieselbe Länge, die Knoten sind gleichmäßig zu verteilen und die zugrunde liegende Struktur wird widergespiegelt.

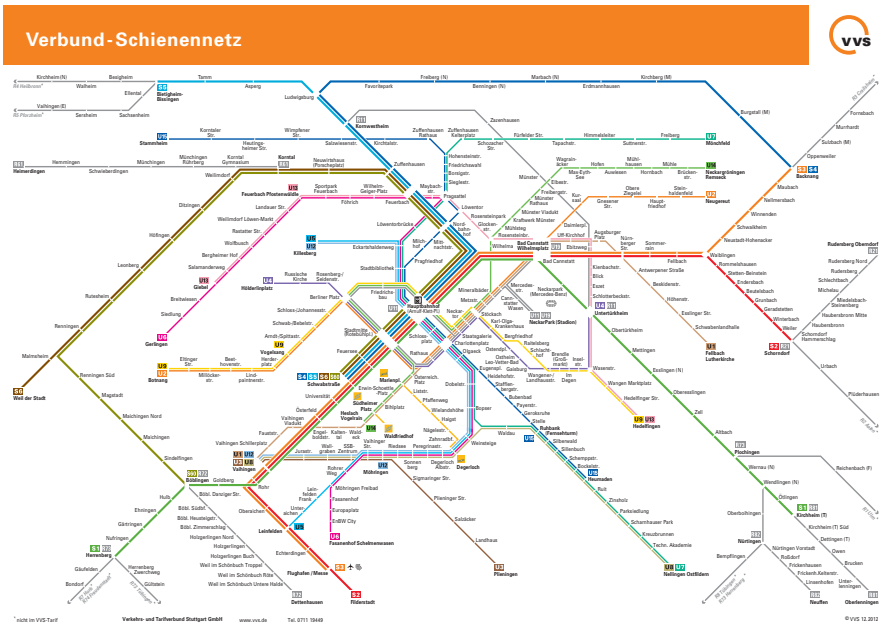


Abbildung 2.7.: Ansicht des Liniennetzplans des VVS [Ver13].

2.7.2. Graphen Layout

Bei der Darstellung von Graphen gibt es unterschiedliche Ansätze diese zu strukturieren. Die Ansätze unterscheiden sich auch in ihrer Eignung für bestimmte Anwendungsgebiete, daher sollte je nach Anwendungsgebiet ein geeignetes Layout gewählt werden.

Hierarchisches Layout

Bei dieser Struktur wird versucht die gegebene Hierarchie in der Visualisierung widerzuspiegeln. Das hierarchische Layout eignet sich daher gut zur Abbildung von Hierarchien. Funktionsbäumen, Syntaxbäume, Bäume oder Geschäftsprozesse werden oft hierarchisch abgebildet. Meist werden gerichtete Kanten verwendet und oft zeigen die meisten Kanten in die gleiche Richtung und oft wird auch ihre Überschneidung minimiert.
Bäume können als eine Spezialisierung des hierarchischen Layouts angesehen werden. Mittels Bäumen können Monohierarchien (d.h.: die Existenz einer Wurzel ist gegeben) gut dargestellt werden, Abbildung 2.8 visualisiert eine Monohierarchie mittels hierarchischem Layout.

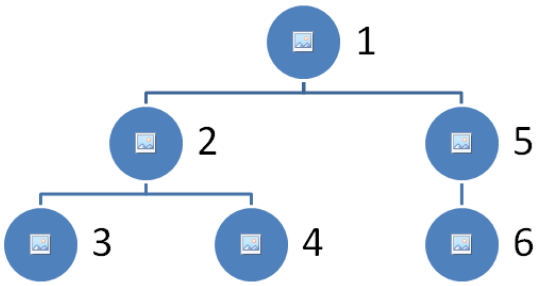


Abbildung 2.8.: Beispiel eines hierarchischen Graphen.

Zirkuläres Layout

Ein zirkulären Layouts spiegelt Gruppenstrukturen wieder. Die Anordnung der Knoten innerhalb eines zirkuläres Layout entspricht der Verbindungsstrukturen der Knoten. Dieses Layout eignet sich gut zur Analyse von Netzwerken, seien sie technischer oder sozialer Natur [Ten13]. Abbildung 2.9 demonstriert ein zirkuläres Layout anhand eines Beispiels.

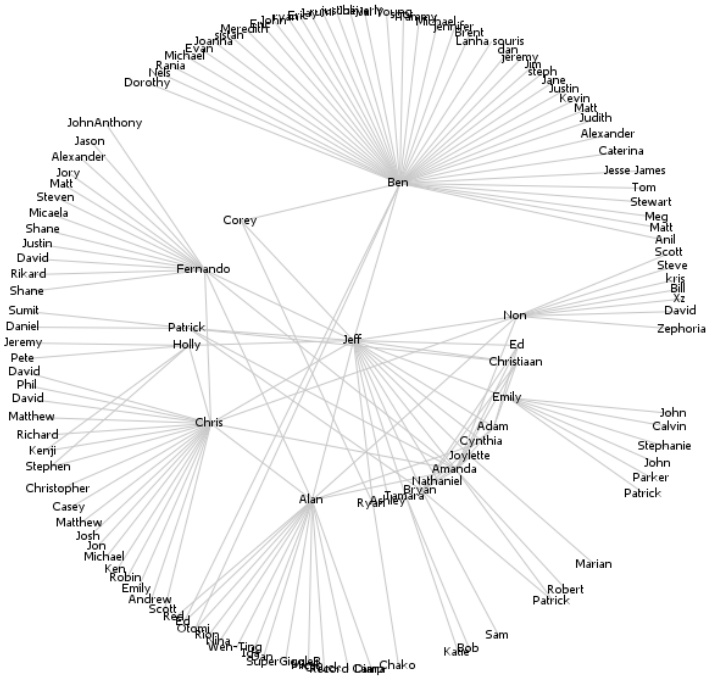


Abbildung 2.9.: Beispiel eines zirkulären Layouts, entnommen aus der Prefuse Demo [Ber13, RadialGraphView].

Layout des längsten Pfad

Beim Layout des längsten Pfades wird der Pfad mit den meisten Knoten zwischen allen Knoten ohne Vorgänger und Knoten ohne Nachfolger ermittelt. Dieser Pfad wird zur Ausrichtung des gesamten Graphen verwendet. Alle Knoten des längsten Pfades werden entlang einer Geraden ausgerichtet, alle anderen Knoten werden um diese Gerade angeordnet. Diese Art der Darstellung eignet sich vor allem für ereignisgesteuerte Prozessketten (d.h.: zur Darstellung von Geschäftsprozessen). Auch Unified Modeling Language (UML) und Business Process Model and Notation (BPMN) Diagramme können auf diese Art und Weise übersichtlich dargestellt werden.

Kräftebasiertes Layout

> „Der Energie-Layout-Algorithmus nutzt einen kräfte-ausgleichenden Layout-Algorithmus für die Anordnung der Knoten im Graphen. Dieser Algorithmus betrachtet den Graphen als ein Kräftesystem und strebt eine möglichst energiearme Anordnung an. In diesem System werden die Knoten eines Graphen als elektrisch geladene Teilchen mit Abstoßungskräften und die Kanten als Federn mit Rückhaltefunktion betrachtet. Das Resultat des Energie-Layouts ist besonders natürlich und ideal für das Layout von Sozialen Netzen und für die Simulation von chemischen oder physikalischen Modellen. Es erzeugt ein harmonisches und ausbalanciertes Ergebnis, obwohl sich hier Kanten überschneiden können.“ [Ten13]

Kräftebasierte Layouts werden oft für geradlinige Zeichnungen und ungerichtete Graphen eingesetzt, denn sie sind populär und ergeben eine übersichtliche und einfach zu verstehende Darstellung. Das Resultat eines kräftebasierten Layoutalgorithmus ist ein System mit minimaler Energie, nicht jedoch zwangsweise das System mit der minimalsten Energie [Sch13]. Ein Beispiel eines kräftebasierten Layouts ist in Abbildung 2.10 abgebildet. Kräftebasierte Layouts sind leicht adaptierbar, konfigurierbar, robust und skalierbar. Sie haben weder Qualitätsgarantien noch Laufzeitgarantien [TM06]. Kräftebasierte Algorithmen können beispielsweise auch zur Darstellung eines Entity-Relationship Model (ERM) verwendet werden.

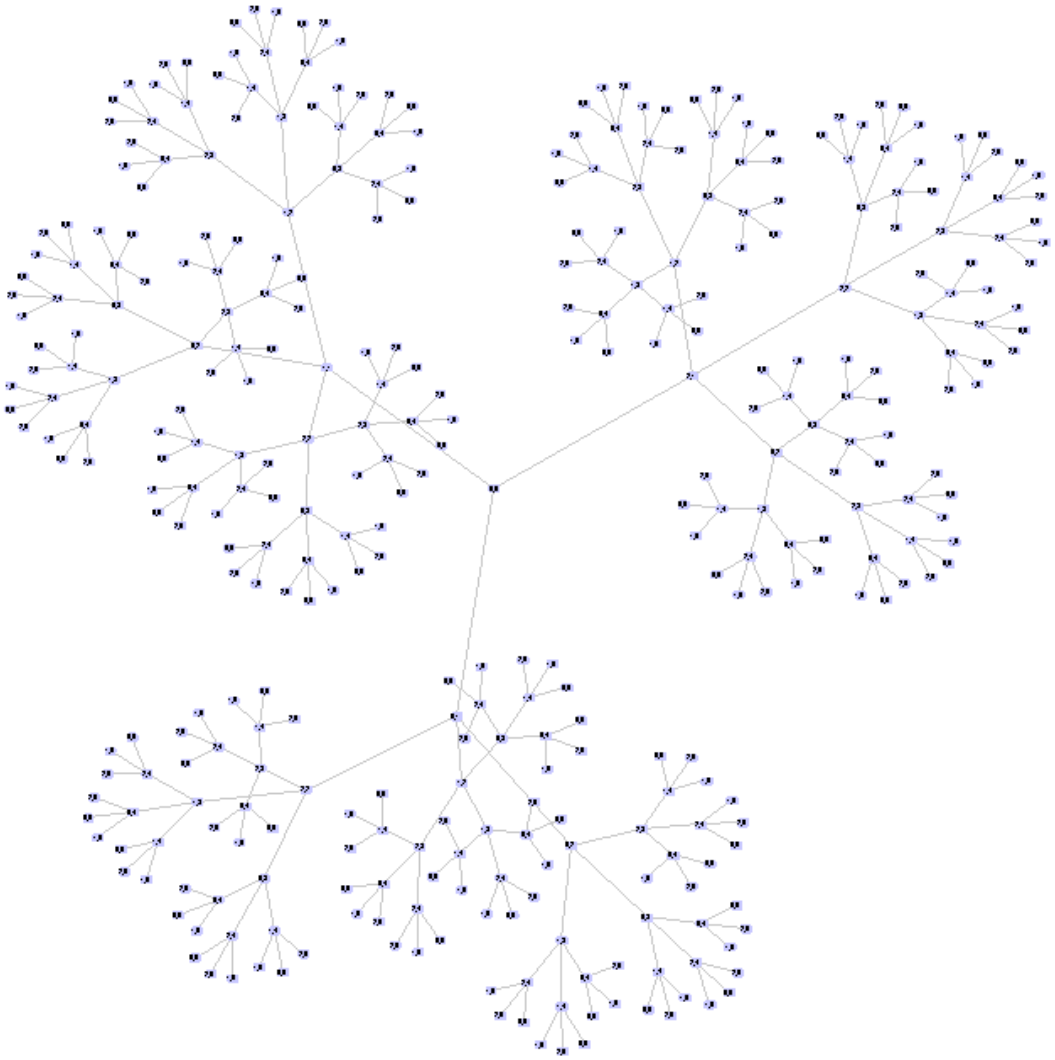


Abbildung 2.10.: Darstellung eines kräftebasierte Algorithmus aus der Prefuse Demonstration [Ber13].

3. Themenverwandte Arbeiten

In diesem Abschnitt werden themenverwandte Arbeiten, die sich mit der Visualisierung von Ontologien befassen, vorgestellt und ihre Vorteile und Nachteile gegenüber dem in Kapitel 4 vorgestellten Konzepts diskutiert.

3.1. GrOWL

GrOWL ist ein 2005 veröffentlichtes Werkzeug zur Visualisierung und Bearbeitung von Ontologien auf Basis von OWL-Ontologien und bzw. Ontologien auf Basis der Beschreibungslogik (DL) [KWV07]. Wie Protégé ist GrOWL ein Editor zum Bearbeiten von Ontologien, Abbildung 3.1 verdeutlicht die Möglichkeiten Ontologien zu bearbeiten. Eine Visualisierung der bereits in Abschnitt 2.4 vorgestellten und mittels OntoGraf visualisierten MUTO-Ontologie ist in Abbildung 3.1 dargestellt.
GrOWL wird von der Universität von Vermont nicht mehr zur Verfügung gestellt, Besucher der Webseite erhalten stattdessen eine Fehlermeldung[1]. Mithilfe des Internet-Archiv[2] kann die Seite des Projekts aus dem Jahre 2008 weiterhin aufgerufen werden[3]. Die Darstellungen aus Abbildung 3.1 entstammen der Standalone Edition, welche die Versionsnummer `0.02` trägt. Die im Internet-Archiv weiterhin aufrufbare Seite stellt zwei Versionen des Tools vor: Die ältere Version verwendet die OWL-API, während die neuere Version Prefuse und JENA verwendet. Bei JENA[4] handelt es sich um ein Framework zum Einlesen und Schreiben von Ontologien im RDF-Format. JENA ist damit der OWL-API aus Abschnitt 2.5 ähnlich und unterstützt beispielsweise SPARQL[5]. Dabei handelt es sich um eine graphbasierte, semantische Abfragesprache für RDF, SPARQL ist zugleich ein Akronym für „SPARQL Protocol And RDF Query Language“. Die im Internet-Archiv verfügbare Version 0.02 von GrOWL akzeptiert Ontologien im RDF-Format, andere Formate können nicht eingelesen werden. Des Weiteren stürzt die im Internet-Archiv verfügbare Version 0.02 beim Einlesen der MUTO-Ontologie öfters ohne Fehlermeldung ab. Damit ist eine zuverlässige Erstellung und Bearbeitung von Ontologien mit dieser Version von GrOWL nicht gewährleistet. Über GrOWL können Zusatzinformationen zu Literalen und Klassen abgerufen werden. Zusätzliche Informationen über Kanten werden hingegen nicht dargestellt, auch wenn Kanten wie Knoten markiert werden

[1] http://www.uvm.edu/~skrivov/GrOWLEditor.jar
[2] http://archive.org/about/contact.php
[3] http://web.archive.org/web/20080424033752/http://www.uvm.edu/~skrivov/growl/index.html
[4] http://jena.apache.org
[5] http://www.w3.org/2009/sparql/wiki/Main_Page

können. Rechts wird, wie Abbildung 3.1 zeigt, eine Übersicht aller enthaltenen Klassen gegeben.
Innerhalb von GrOWL werden `Object Properties` blau dargestellt, während `Datatype Properties` mit einer gelben Farbe visualisiert werden. Der Namespace eines Elements wird innerhalb des Knoten-Kanten-Diagramms nicht angezeigt. Zugriff auf die verwendeten Namespaces erhält der Benutzer in Tabellenform, nach Betätigen der `View Metadata` Schaltfläche. Klassen werden in GrOWL ovalförmig, während Literale quadratisch dargestellt werden. `SubClassOf` Beziehungen werden in GrOWL durch einen durchgezogenen Pfeil dargestellt. Äquivalenzen werden durch einen Mesomeriepfeil repräsentiert, dabei handelt es sich um einen Pfeil, der zwei Spitzen besitzt. GrOWL orientiert sich bei der Darstellung von OWL-Konstrukten an prädikatenlogische Ausdrücke und den DL-Klassenkonstrukte. Ein Auszug hiervon ist in Abbildung 3.2 dargestellt. Instanzen werden in GrOWL durch ein nicht gefülltes blaues Quadrat repräsentiert. Das in Abschnitt 2.1 vorgestellte Beispiel aus Abbildung 2.1 wird in GrOWL gemäß Abbildung 3.3 visualisiert.
Eine Besonderheit von GrOWL besteht in der Möglichkeit, Operatoren zusammengefasst darzustellen, dass in Abbildung 3.4 verdeutlicht wird.
Der Benutzer kann Instanzen ausblenden, ein Überblick über die Anzahl der Instanzen wird jedoch nicht gegeben. Durch die Darstellung sämtlicher Instanzen und Restriktionen werden große Ontologien schnell unübersichtlich, was bei der Visualisierung der WINE-Ontologie[6] in GrOWL deutlich zum Vorschein kommt und in Abbildung 3.5 veranschaulicht wird.
GrOWL kann verschiedene Einschränkungen der Properties, visuell darstellen, hierzu zählen Werteeinschränkungen, Einschränkungen hinsichtlich der Kardinalität sowie die OWL-Einschränkungen `someValuesFrom` und `allValuesFrom`. Allerdings besitzt GrOWL keine spezielle Visualisierung der unterschiedlichen Property-Typen. Zwar werden `Object Properties` und `Datatype Properties` in unterschiedlichen Farben dargestellt, eine visuelle Repräsentation der Eigenschaften `inverseOf`, `TransitiveProperty`, `SymmetricProperty`, `FunctionalProperty` und `InverseFunctionalProperty` ist hingegen nicht vorgesehen. Im Beispiel aus Abbildung 3.1 werden die funktionalen Properties `nextTag` und `previousTag` exakt wie `meaningOf` dargestellt. Bei `meaningOf` handelt es sich um eine Property, die keine Domain-Angabe besitzt. Ihre Beschreibung im RDF-Format ist in Quelltext 3.1 gegeben. Wie Abbildung 3.6 zeigt, kann diese Information lediglich über die rechte Informationsleiste abgerufen werden. Falls der Benutzer herausfinden möchte, welche Property symmetrisch ist, so muss er jede Einzelne markieren und die rechte Informationsleiste überprüfen.
GrOWL stellt damit nur einen Teilaspekt der Ontologie dar und eignet sich daher nicht für die ganzheitliche Visualisierung.

[6] http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine

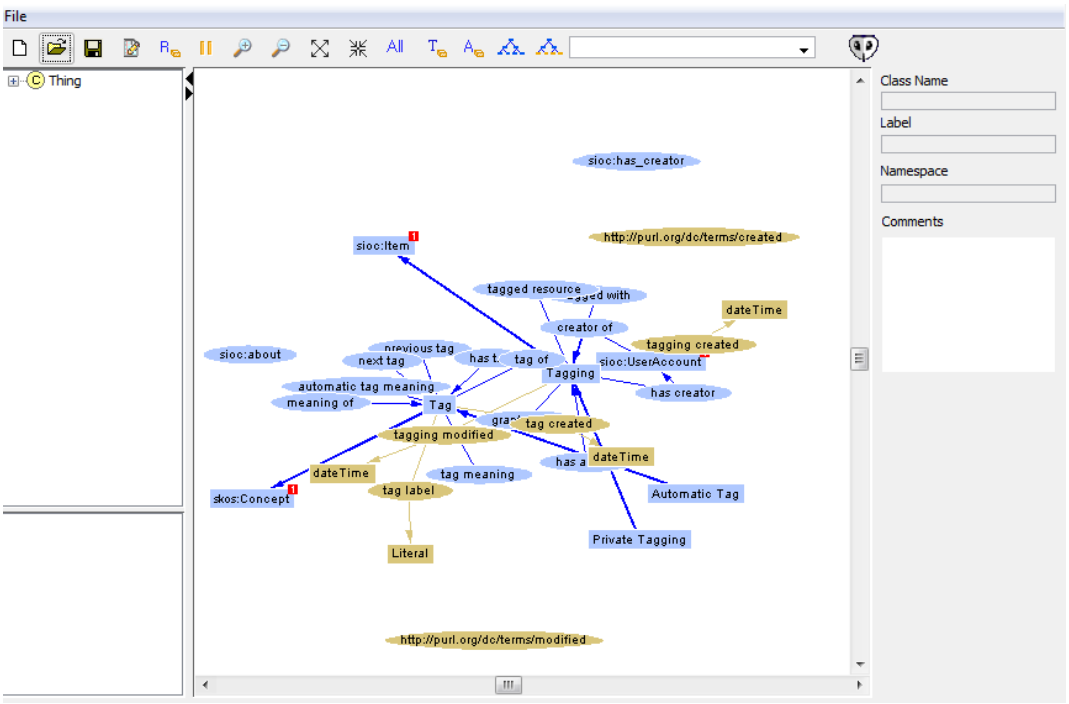


Abbildung 3.1.: Darstellung der MUTO-Ontologie mittels GrOWL.

Definition of class C	**The diagram G(C)**	**Base node BN(C)**
Named Class **C**	C	C
Intersection $C_1 \sqcap C_2$	BN(C1) ⊓ BN(C2)	⊓
Union $C_1 \sqcup C_2$	BN(C1) ⊔ BN(C2)	⊔
Complement $\neg C_1$	¬ BN(C1)	¬
Enumeration $\{o_1, o_2\}$	∈ o2 o1	∈
Exist Restriction $\exists R.C_1$	∃:R BN(C1)	∃:R
For all Restriction $\forall R.C_1$	∀:R BN(C1)	∀:R
Number Restriction $\geq nR$	Eg. ≥5:R	≥5:R
Number Restriction $\leq n\,R$	Eg. ≤7:R	≤7:R
Value Restriction $R:o$	R o	R

Abbildung 3.2.: Konstrukte in GrOWL, entnommen aus [KWV07].

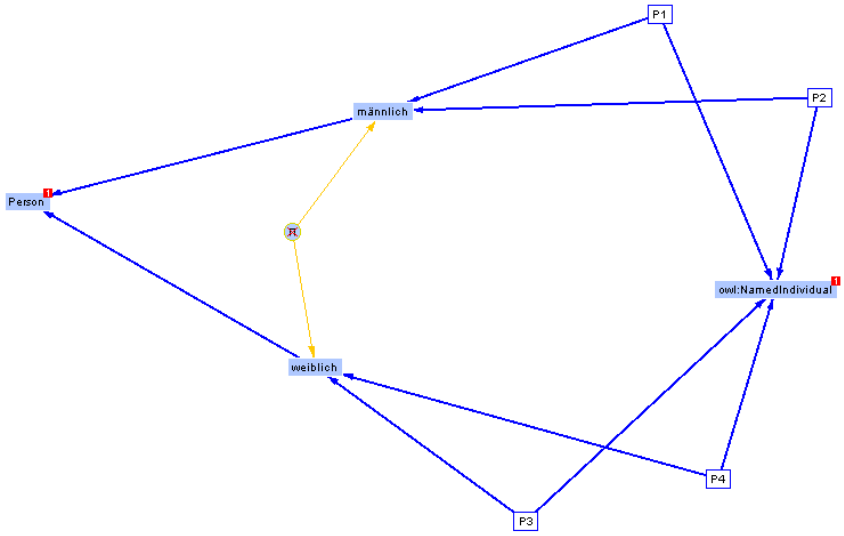


Abbildung 3.3.: Instanzen werden in GrOWL visualisiert.

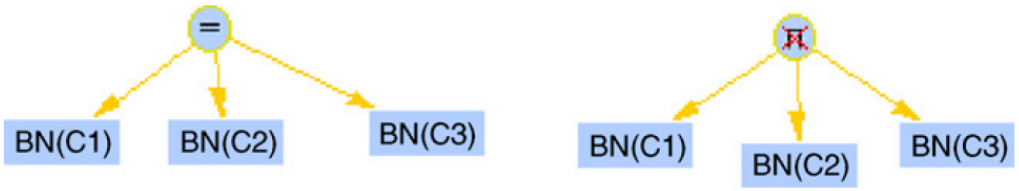


Abbildung 3.4.: Operatoren können zusammengefasst werden.

Quelltext 3.1 Property `meaning of` aus der MUTO-Ontologie.

```
<owl:ObjectProperty rdf:about="http://purl.org/muto/core#meaningOf">
   <rdfs:label xml:lang="en">meaning of</rdfs:label>
   <rdfs:comment xml:lang="en">The number of tags that can be linked to one and the same
       meaning is theoretically unlimited.</rdfs:comment>
   <rdfs:range rdf:resource="http://purl.org/muto/core#Tag"/>
   <owl:inverseOf rdf:resource="http://purl.org/muto/core#tagMeaning"/>
      <rdfs:isDefinedBy rdf:resource="http://purl.org/muto/core#"/>
 </owl:ObjectProperty>
```

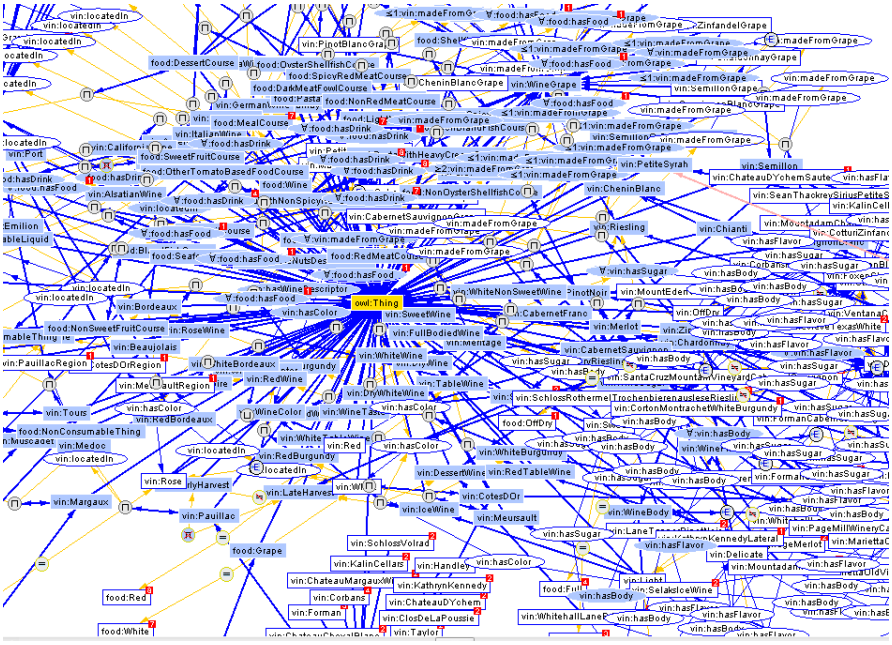


Abbildung 3.5.: Darstellung der WINE-Ontologie mittels GrOWL.

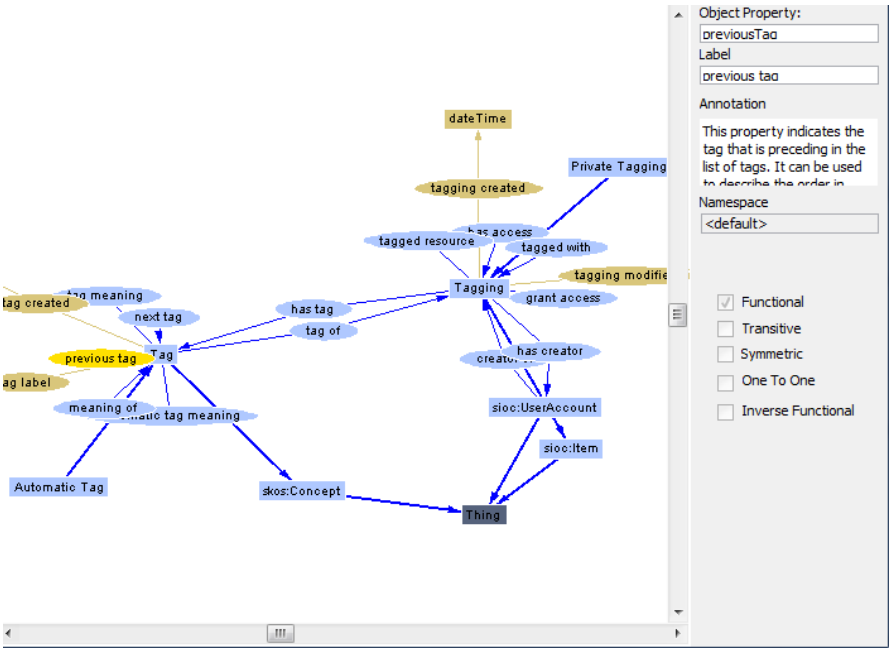


Abbildung 3.6.: Visualisierung der `meaning of` Property aus der MUTO-Ontologie.

3.2. SOVA

SOVA ist ein Protégé Plug-in zur Visualisierung von Ontologien, ausgeschrieben steht SOVA für „Simple Ontology Visualization API“. Entwickelt wurde es von Piotr Kunowski und Tomasz Boiński von der Gdańsk-Universität der Technik[7], die ihre erste Version von SOVA im Jahre 2010 veröffentlichten. [PK12]. Eine Visualisierung der MUTO-Ontologie mittels SOVA ist in Abbildung 3.7 dargestellt. Ein Verzeichnis aller Symbole samt entsprechender OWL-Statements steht online unter [PK10] zur Verfügung.
Die Ansicht innerhalb des SOVA Plug-ins ist dreigeteilt. Links erhält der Benutzer eine Übersicht über die Hierarchie aller enthaltenen Klassen. Diese Ansicht wird von Protégé zur Verfügung gestellt, der Anwender könnte diese Ansicht anderen Visualisierungen ebenfalls hinzufügen. In der Mitte befindet sich die eigentliche Visualisierung. Rechts werden zusätzliche Informationen angezeigt, sofern der Benutzer eine Kante anklickt. Ein Beispiel dieser dreiteiligen Ansicht ist in Abbildung 3.7 gegeben. Anstelle des mittigen Graphen kann auch eine weitere Hierarchie der Klassen und Individuen eingeblendet werden, um dies zu bewerkstelligen, wird der `hermit owl reasoner` benötigt[8].
SOVA verwendet ein Spring-Layout, dies hat zur Folge, dass sich Knoten gegenseitig abstoßen, während Kanten Anziehungskräfte modellieren. Durch diese Layoutform werden semantisch ähnlichere Knoten dichter als semantisch unähnlichere Knoten platziert. Innerhalb der Visualisierung der Ontologie verwendet SOVA ähnliche Symbole wie das bereits in Abschnitt 3.1 vorgestellte GrOWL. Instanzen werden als grau gefüllte Quadrate den jeweiligen Klassen zugeordnet. Eine Visualisierung des Beispiels aus Abschnitt 2.1 mittels SOVA wurde bereits in Abbildung 2.4 gezeigt. Eine Information über die Anzahl der enthaltenen Instanzen wird nicht gegeben, der Benutzer muss diese Information durch Zählen der abgebildeten Instanzen selbst ermitteln. Im Gegensatz zu GrOWL aus Abschnitt 3.1 besteht keine Möglichkeit des Ausblendens der Instanzen. Analog zu GrOWL können in SOVA nur Knoten angeklickt werden. Das Auswählen von Kanten ist nicht möglich. Fährt der Benutzer mit dem Mauszeiger über einen Knoten und damit auf eine Klasse bzw. ein Literal, so werden alle damit verbundenen Knoten farblich hervorgehoben. Des Weiteren ist dies die einzige Möglichkeit den Namespace der entsprechenden Klasse bzw. Literal in Erfahrung zu bringen. SOVA bietet dem Nutzer die Möglichkeit nach dem Namen einer Klasse bzw. eines Literals zu suchen, die Verwendung von regulären Ausdrücken ist nicht möglich. Abbildung 3.7 zeigt sowohl das farbliche Hervorheben durch eine weiße Farbe, die Suche sowie das Auslesen des Namespace der angeklickten Klasse. Abbildung 3.7 zeigt allerdings auch einen Fehler des SOVA Plug-ins. Quelltext 3.2 zeigt die `Object Property tagOf`. Sie enthält sowohl einen Kommentar als auch ein Label. Beides wird innerhalb des Plug-ins nicht angezeigt. Dies kann sich irritierend auf Benutzer auswirken, schließlich erweckt Quelltext 3.2 den Eindruck, dass die entsprechende Property wirklich keine weiteren Angaben enthalten würde.
SOVA verwendet bei der Visualisierung von Kardinalitäten Existenzquantoren. Klassen werden als abgerundetes Quadrat mit der Hintergrundfarbe „steelblue“ dargestellt. `Datatype Properties` und `RDF-Properties` werden ebenfalls als abgerundetes Quadrat abgebildet.

[7] http://www.pg.gda.pl/en/
[8] http://hermit-reasoner.com

`Datatype Properties` haben eine hellgrüne Hintergrundfarbe, während RDF-Properties „darkmagenta“ als Hintergrundfabe besitzen. `Datatype Properties` werden durch das Plug-in nicht repräsentiert. Abbildung 3.7 enthält beispielsweise keine Visualisierung des `Datatype Properties` `tag label`. `Datatype Properties` sind innerhalb der MUTO-Ontologie gemäß Quelltext 3.3 vorhanden.
Ähnlich wie GrWOL visualisiert SOVA äquivalente Klassen mittels Mesomeriepfeilen. Die `disjointWith` Eigenschaft wird mittels invertiertem Mesomeriepfeil dargestellt. Dabei werden beide Pfeilspitzen vertikal gespiegelt. SOVA beherrscht ebenfalls das zusammenfassen von Relationen, beispielsweise muss nicht jedes Individuum einer `oneOf` Relation ein eigenes Symbol besitzen. Eine Visualisierung dieser `oneOf` Relation gemäß [PK10] wird in Abbildung 3.8 gezeigt.
Im Gegensatz zu dem in Abschnitt 3.1 vorgestelltem GrOWL ist SOVA in der Lage, die Eigenschaften `inverseOf`, `TransitiveProperty`, `SymmetricProperty`, `FunctionalProperty` und `InverseFunctionalProperty` gemäß Abbildung 3.9 visuell darzustellen. SOVA stellt `unionOf`, `intersectionOf` und `complementOf` mit denselben Symbolen wie GrOWL aus Abbildung 3.2 dar. SOVA verwendet allerdings andere Pfeilspitzen und die daraus resultierende Klasse wird für die weitere Verwendung ebenfalls grafisch dargestellt. Ein Beispiel für `unionOf` ist in Abbildung 3.10 abgebildet. Kardinalitäten werden von SOVA ebenfalls unterstützt, diese sehen für unerfahrene Anwender jedoch recht merkwürdig aus. Eine derartige Angabe ist in Abbildung 3.11 vorhanden.
Durch die Visualisierung aller Instanzen und durch Repräsentation von OWL-Konstrukten mittels einer Vielzahl von geometrischen Formen werden große Ontologien schnell unübersichtlich. Abbildung 3.12 stellen eine Visualisierung der WINE-Ontologie dar. SOVA verwendet ein kräftebasiertes Layout, durch das die wichtigsten Bestandteile der Ontologie mittig platziert werden.
Aufgrund der fehlenden `Datatype Properties` stellt SOVA nur einen Teilaspekt der Ontologie dar und eignet sich daher nicht für die ganzheitliche Visualisierung von Ontologien.

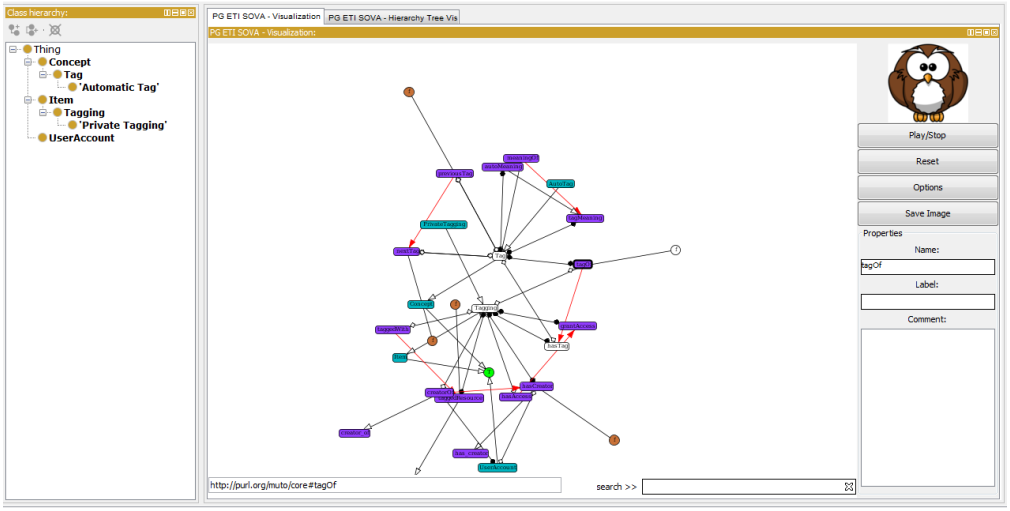


Abbildung 3.7.: Das SOVA Plug-in bei geladener MUTO-Ontologie.

Quelltext 3.2 `tagOf Property` aus der MUTO-Ontologie.

```
<owl:ObjectProperty rdf:about="http://purl.org/muto/core#tagOf">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:label xml:lang="en">tag of</rdfs:label>
    <rdfs:comment xml:lang="en">Every tag is linked to exactly one tagging. This results from
        the fact that tags with same labels are NOT merged in the ontology.</rdfs:comment>
    <rdfs:domain rdf:resource="http://purl.org/muto/core#Tag"/>
    <rdfs:range rdf:resource="http://purl.org/muto/core#Tagging"/>
    <owl:inverseOf rdf:resource="http://purl.org/muto/core#hasTag"/>
        <rdfs:isDefinedBy rdf:resource="http://purl.org/muto/core#"/>
</owl:ObjectProperty>
```

Quelltext 3.3 `tagLabel Property` aus der MUTO-Ontologie.

```
<owl:DatatypeProperty rdf:about="http://purl.org/muto/core#tagLabel">
       rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:label xml:lang="en">tag label</rdfs:label>
    <rdfs:comment xml:lang="en">Every tag has exactly one label (usually the one given by the
        user) - otherwise it is not a tag. Additional labels can be defined in the resource
        that is linked via muto:tagMeaning.</rdfs:comment>
    <owl:versionInfo>Version 1.0: The subproperty relation to rdfs:label has been removed for
        OWL DL conformance (rdfs:label is an annotation property and one cannot define
        subproperties for annotation properties in OWL DL).</owl:versionInfo>
    <rdfs:domain rdf:resource="http://purl.org/muto/core#Tag"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
        <rdfs:isDefinedBy rdf:resource="http://purl.org/muto/core#"/>
</owl:DatatypeProperty>
```

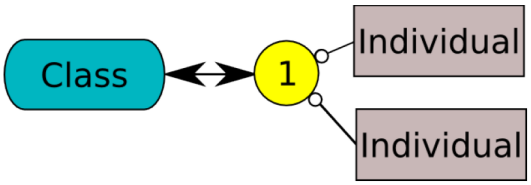


Abbildung 3.8.: Visualisierung der `oneOf` Property in SOVA, entnommen aus [PK10].

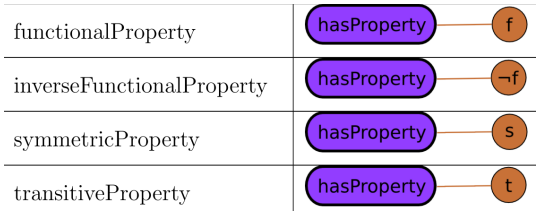


Abbildung 3.9.: Darstellung verschiedener Properties, entnommen aus [PK10].

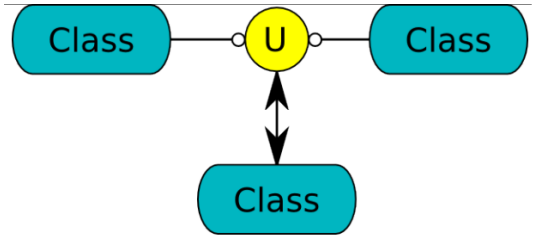


Abbildung 3.10.: Darstellung von `unionOf`, entnommen aus [PK10].

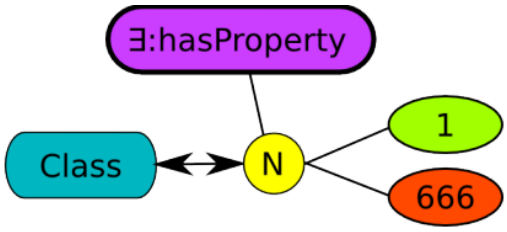


Abbildung 3.11.: Kardinalität in SOVA, entnommen aus [PK10].

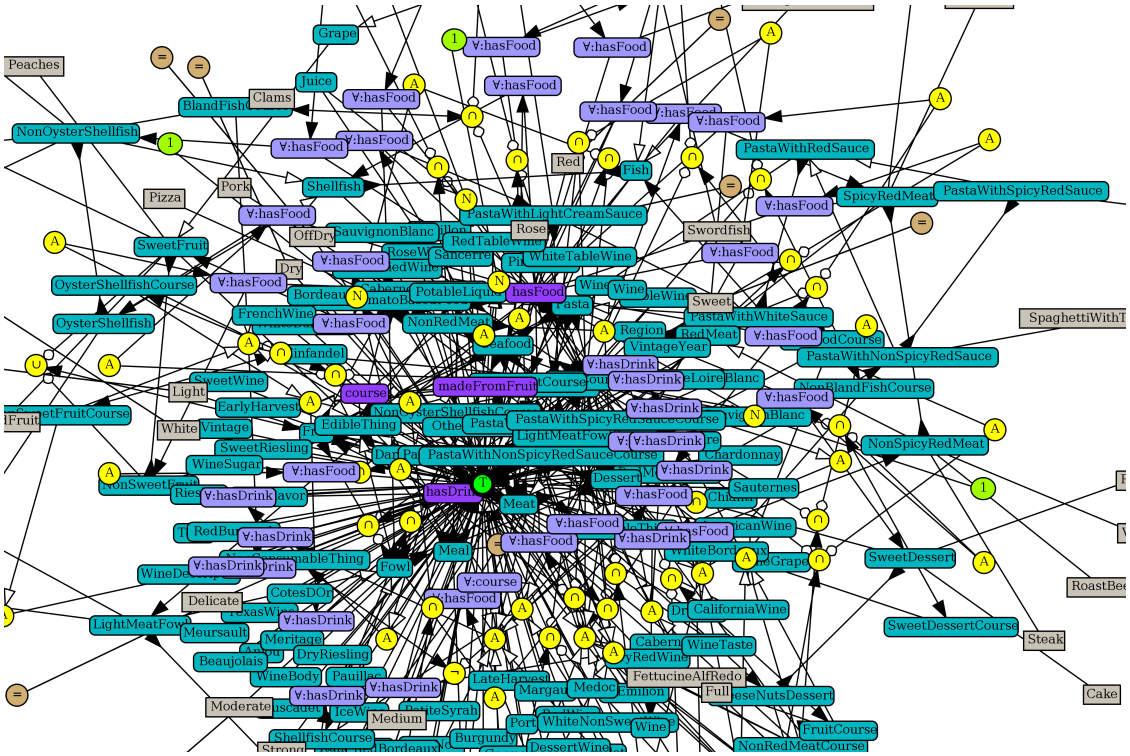


Abbildung 3.12.: Visualisierung der WINE-Ontologie mittels SOVA.

3.3. OWLViz

Bei OWLViz handelt es sich um ein Protégé Plug-in zur Visualisierung von Ontologien und ist in der Standardinstallation von Protégé enthalten[9]. Eine Visualisierung des Beispiels aus Abschnitt 2.1 wurde bereits in Abbildung 2.2 vorgestellt. In Abbildung 3.13 findet sich eine Darstellung der mittels OWLViz visualisierten MUTO-Ontologie. [LDA11]. OWLViz verwendet das, in dem späteren Abschnitt 5.1 näher beschriebene, Grafikframework GraphViz zur Generierung der Visualisierung. Dies ist eine Ursache für die eher statisch gehaltene Visualisierung von OWLViz. Dies hat zur Folge, dass der Benutzer Knoten des Knoten-Kanten-Diagramms zwar markieren, jedoch diese nicht mittels drag & drop verschieben kann.
Wie bereits in Abbildung 2.2 gezeigt, visualisiert OWLViz keine Instanzen. Die Visualisierung der MUTO-Ontologie aus Abbildung 3.13 demonstriert, dass OWLViz auch keine Properties visuell darstellen kann. Lediglich die `SubClassOf` Beziehung zwischen Klassen wird dargestellt. Somit stellt OWLViz nur einen sehr kleinen Ausschnitt aus OWL grafisch dar und eignet sich daher nicht die ganzheitliche Visualisierung von Ontologien.

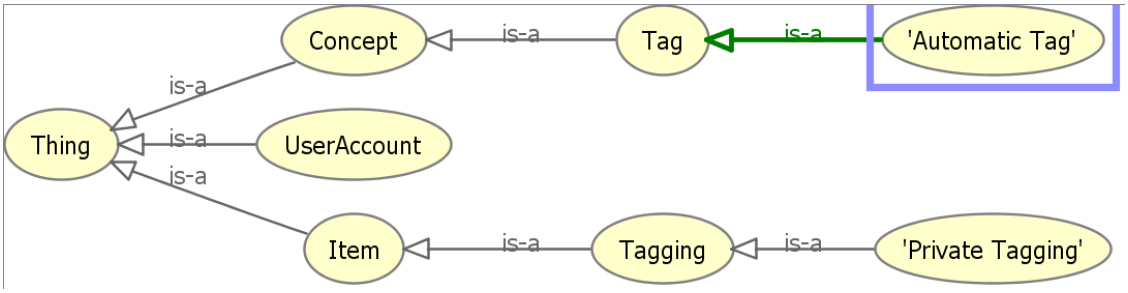


Abbildung 3.13.: Visualisierung der MUTO-Ontologie mittels OWLViz.

3.4. OntoGraf

Bei OntoGraf handelt es sich, wie bei OWLViz aus Abschnitt 3.3, um ein Protégé Plug-in zur Visualisierung von Ontologien und ist ebenfalls in der Standardinstallation von Protégé enthalten[10]. Wie Abbildung 3.14 zeigt, stellt OntoGraf nicht nur Klassen visuell dar, sondern auch Properties und Instanzen. Knoten stehen für Klasse und Instanzen, während Kanten die grafische Repräsentation von Properties darstellen. Die Abbildung von Instanzen wurde bereits in Abbildung 2.3 vorgestellt.
Innerhalb von OntoGraf kann der Benutzer Knoten markieren und diese verschieben, während Kanten nur markiert werden können. Bei beiden Elementen erhält der Benutzer Tooltips, was in Abbildung 3.14 für Kanten und in Abbildung 3.15 für Knoten veranschaulicht wird. In beiden Fällen werden die, mit dem entsprechenden Element, verbundenen Objekte grafisch hervorgehoben. Zwar sind die Tooltips für Knoten umfangreich, die entsprechenden

[9] http://protegewiki.stanford.edu/wiki/OWLViz
[10] http://protegewiki.stanford.edu/wiki/OntoGraf

OWL-Einträge werden allerdings nicht aufbereitet. Stattdessen erhält der Benutzer eine, durch Absätze formatierte, Auflistung aller zugehörigen Annotationen. Ein Beispiel für einen derartigen Tooltip ist in Abbildung 3.15 abgebildet. Tooltips für Kanten verhalten sich analog, ein Beispiel hierfür wird in Abbildung 3.14 gegeben.
Innerhalb von OntoGraf kann der Benutzer Knoten markieren und diese verschieben, während Kanten nur markiert werden können. Bei beiden Elementen erhält der Benutzer Tooltips, was in Abbildung 3.14 für Kanten und in Abbildung 3.15 für Knoten veranschaulicht wird. In beiden Fällen werden die, mit dem entsprechenden Element, verbundenen Objekte grafisch hervorgehoben. Zwar sind die Tooltips für Knoten umfangreich, die entsprechenden
OWL-Einträge werden allerdings nicht aufbereitet. Stattdessen erhält der Benutzer eine, durch Absätze formatierte, Auflistung aller zugehörigen Annotationen. Ein Beispiel für einen derartigen Tooltip ist in Abbildung 3.15 abgebildet. Tooltips für Kanten verhalten sich analog, ein Beispiel hierfür wird in Abbildung 3.14 gegeben.
Die fehlenden Kantenbeschriftungen stellt für den Nutzer ein Ärgernis dar, denn er muss die Kantenfarbe mit der Property-Übersicht aus Abbildung 3.16 eigenständig abgleichen. Die Kantenfarben werden zufällig gewählt, sie stellen keine grafische Darstellung eines bestimmten Property-Typs dar. Ein und dieselbe Kante wird in Abbildung 3.14 und Abbildung 3.15 unterschiedlich gefärbt, obwohl alle drei Abbildungen die MUTO-Ontologie visuell darstellen. Im Gegensatz zur Farbe ist die Form einer Kante relevant. Durchgezogene Linien stehen für Unterklassen-Beziehungen. `Automatic Tag` ist beispielsweise eine Unterklasse von `Tag`. Gestrichelte Linien stehen hingegen für Properties, eine Unterscheidung zwischen `DatatypeProperty` und `Object Property` findet jedoch nicht statt. Daher kann zwischen `inverseOf`, `TransitiveProperty`, `FunctionalProperty` und `InverseFunctionalProperty` nicht unterschieden werden. Eine `SymmetricProperty` kann durch den zurückgelegten Weg visuell erkannt werden, dies wird in Abbildung 3.14 demonstriert.
OntoGraf kann, wie in Abbildung 3.16 gezeigt, eine Übersicht der verwendeten Kantenfarben und der zugehörigen Properties einblenden. Durch diese Übersicht erfährt der Benutzer lediglich den Namen einer Property, Informationen über ihre Funktion bleiben dem Benutzer damit verborgen. Somit eignet sich OntoGraf nur bedingt für unerfahrene Nutzer, die unformatierte Auflistung aller Klassen-Eigenschaften aus Abbildung 3.15 stellt sie möglicherweise vor Verständnisprobleme. Die Nutzer werden durch fehlende Kantenbeschriftungen nur unzureichend beim Erforschen einer Ontologie unterstützt. Die Property-Übersicht aus Abbildung 3.16 erweist sich für einen unerfahrenen Nutzer nur als eingeschränktes Hilfsmittel. Dieser Nutzergruppe sollten daher eher andere Werkzeuge empfohlen werden.

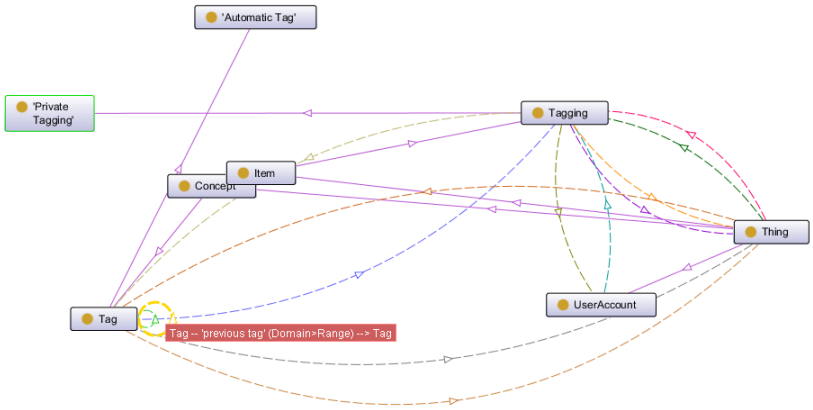


Abbildung 3.14.: OntoGraf: Tooltip bei Properties. Symmetrische Properties werden kreisförmig dargestellt.

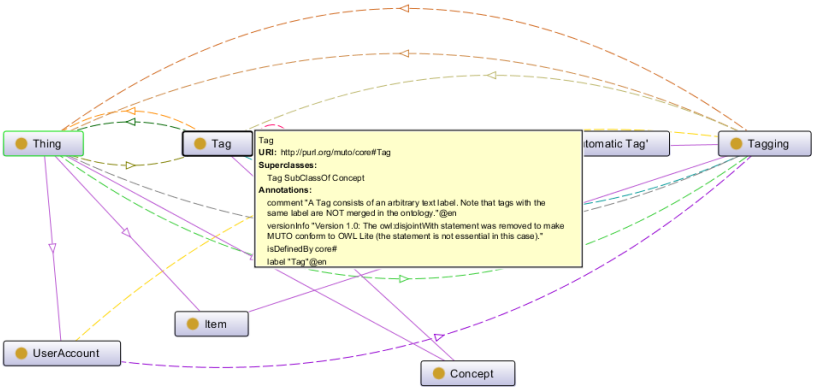


Abbildung 3.15.: OntoGraf: Tooltip bei Klassen.

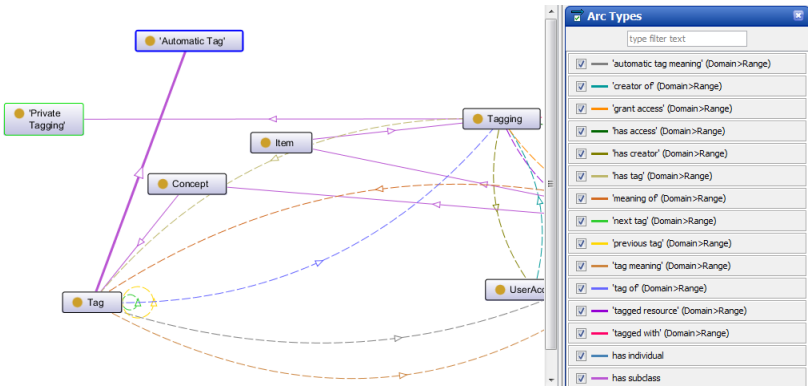


Abbildung 3.16.: OntoGraf: Property-Übersicht.

3.5. TGVizTab

TGVizTab ist ein Plug-in für Protégé 3 zur Generierung visueller Graphen. Es verwendet dabei TouchGraph, eine Java-Umgebung zur Generierung von visuellen Graphen [Ala03]. TGVizTab verwendet ein Spring-Layout, daher stoßen Knoten sich gegenseitig ab, während Kanten Anziehungskräfte für Knoten modellieren. Daher sind semantisch ähnlichere Knoten dichter beisammen abgebildet als semantisch verschiedene Knoten. Abbildung 3.17 zeigt TGVizTab bei der Visualisierung einer Ontologie. TGVizTab verwenden unterschiedliche Farben zur Darstellung von Klassen und Instanzen. Die Wahl der Farben und ihre Sichtbarkeit kann vom Benutzer festgelegt werden.
TGVizTab kann Unterklassenbeziehungen grafisch darstellen, auch Vielfachvererbung sind abbildbar. Die Beziehung zueinander wird jedoch erst visuell dargestellt, wenn der Benutzer mit der Maus auf eine Kante zeigt. Properties werden nicht innerhalb des eigentlichen Graphen, sondern in einem separaten Fenster angezeigt [SA11]. In einer Studie kritisierten Benutzer die spontanen Bewegungen, die in TGVizTab auftreten können. Die Benutzer dieser Studie bevorzugten den Klassenbrowser von Protégé anstelle von TGVizTab [KTH+06].

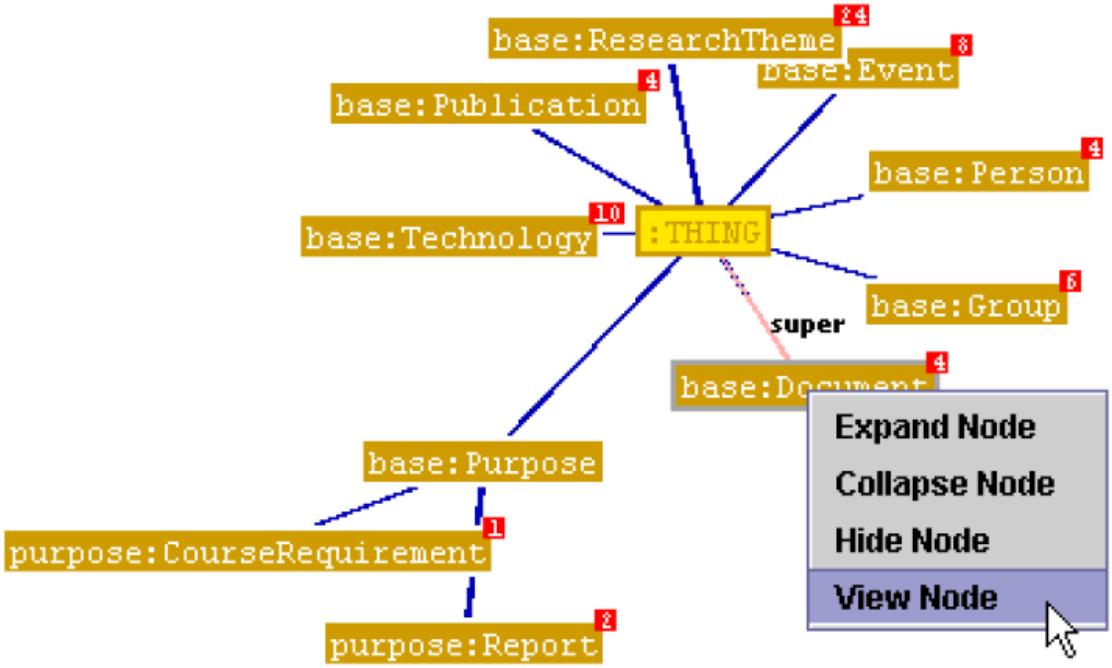


Abbildung 3.17.: Visualisierung mittels TGVizTab, Abbildung entnommen aus [Ala03].

3.6. Jambalaya

Jambalaya ist, wie TGVizTab aus Abschnitt 3.5, ein Plug-in für Protégé 3 zur Visualisierung von Ontologien[11]. Jambalaya verwendet das Grafikframework Piccolo zur Darstellung der Visualisierung. Jambalaya unterstützt unterschiedliche Ansichten zur Visualisierung, so können Ontologien beispielsweise als Treemap oder in Form einer Baumansicht angezeigt werden. Die Baumansicht steht sowohl mit als auch ohne Individuen zur Verfügung. Abbildung 3.18 zeigt die Auswahl der gewünschten Ansicht in Jambalaya. Die Darstellung einer Treemap wird in Abbildung 3.19 gezeigt, während Abbildung 3.20 die Baumansicht mit Individuen und Abbildung 3.21 die Domain & Range Ansicht demonstriert. Wie das bereits in Abschnitt 3.4 vorgestellte OntoGraf stellt Jambalaya sämtliche Properties auf dieselbe Weise dar. Jedes Property wird eine eigene Farbe zugewiesen, eine Unterscheidung in unterschiedliche Property-Typen findet jedoch nicht statt. Abbildung 3.22 zeigt eine, von Jambalaya zur Verfügung gestellte, Übersicht aller verwendeten Property-Farben. Auch über Tooltips erhält der Benutzer keine Information über den jeweiligen Property-Typ. Ein Beispiel hierzu wird in Abbildung 3.21 vorgestellt. In einer Vergleichsstudie schnitt Jambalaya schlechter als der Klassenbrowser von Protégé ab [KTH+06].

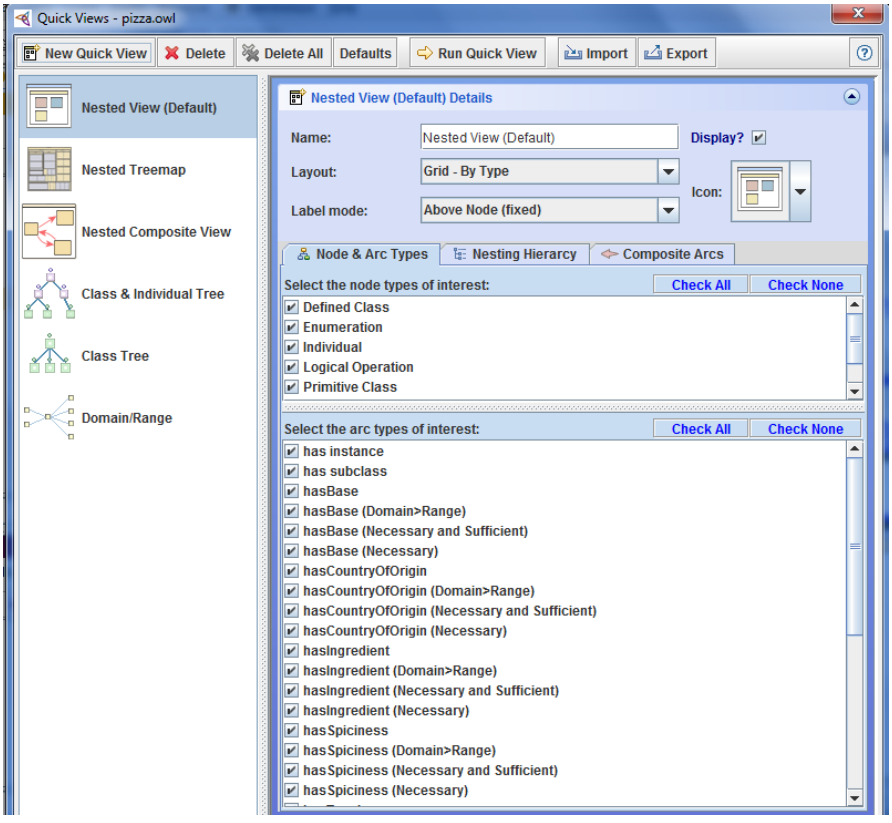


Abbildung 3.18.: Jambalaya bietet verschiedene Ansichten zur Visualisierung.

[11] http://protegewiki.stanford.edu/wiki/Jambalaya

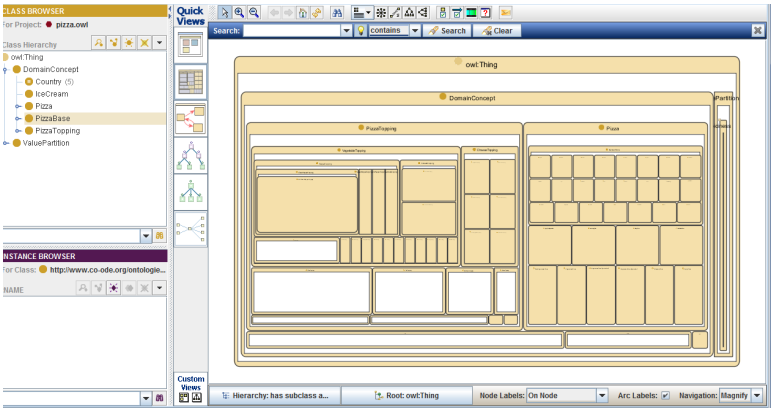


Abbildung 3.19.: Visualisierung mittels TreeMap-Ansicht in Jambalaya.

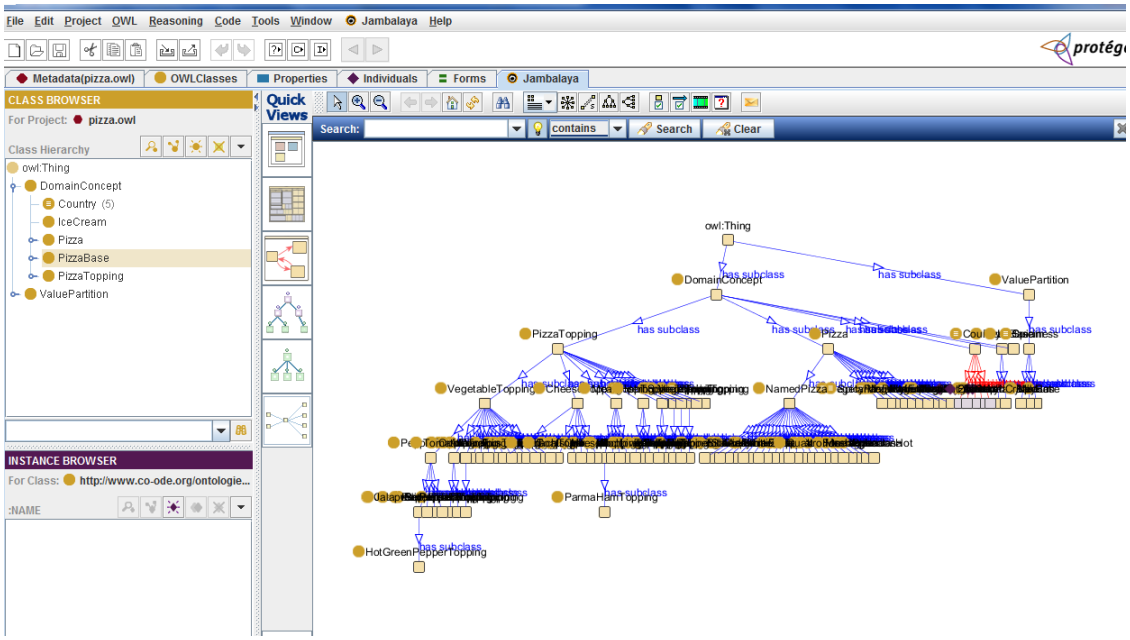


Abbildung 3.20.: Visualisierung mittels ClassTree-Ansicht in Jambalaya.

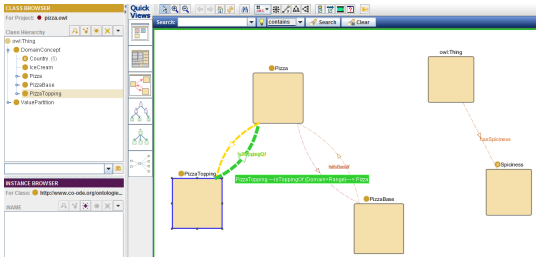


Abbildung 3.21.: Visualisierung mittels Domain- & Range-Ansicht in Jambalaya.

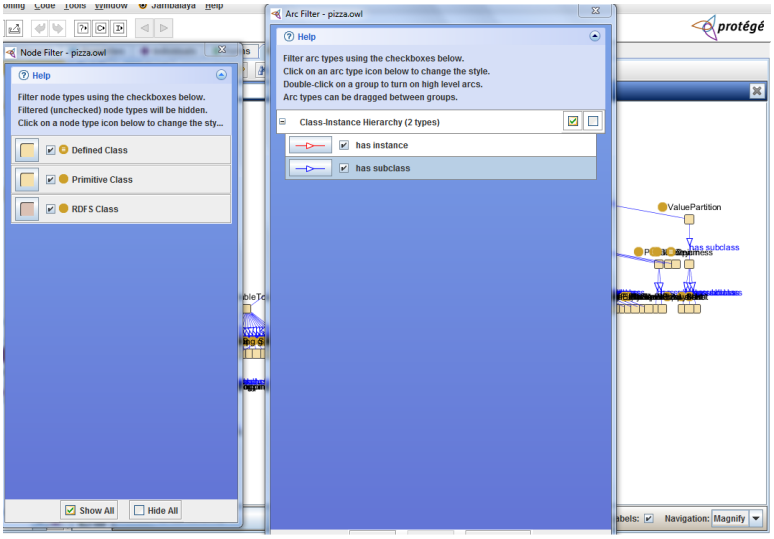


Abbildung 3.22.: Übersicht der verwendeten Farben und Symbole.

3.7. Zusammenfassung

In diesem Kapitel wurden themenverwandten Arbeiten vorgestellt und auf ihre Fähigkeiten zur Visualisierung von Ontologien untersucht. Dabei werden die bestehenden Probleme, Nachteile aber auch die Vorteile dieser Werkzeuge herausgearbeitet.
In OntoGraf (Protégé 4) und Jambalaya (Protégé 3) findet beispielsweise keine Unterscheidung verschiedenartiger Properties statt. So werden verschiedene Properties zwar in unterschiedlichen Farben dargestellt, eine visuelle Unterscheidung zwischen einer `FunctionalProperty` und `TransitiveProperty` oder zwischen `Datatype Property` und `Object Property` ist hingegen nicht möglich. In OntoGraf wird nicht einmal der Name einer Property grafisch dargestellt. TGVizTab für Protégé 3 ermöglicht ebenfalls keine visuelle Unterscheidung verschiedener Property-Typen. OWLViz für Protégé 4 stellt nur Klassen- und Unterklassen-Beziehungen dar. Die Protégé 4 Visualisierungen GrOWL und SOVA haben diesen Mangel nicht. SOVA leidet aber unter Mängeln bei der Darstellung von Kommentaren und Labels und sieht keine Visualisierung für Literale vor. GrOWL visualisiert Literale, hat dafür Mängel bei der Visualisierung der Property-Typen `inverseOf`, `TransitiveProperty`, `SymmetricProperty`, `FunctionalProperty` und `InverseFunctionalProperty`. GrOWL und SOVA orientieren sich bei der Darstellung an prädikatenlogische Ausdrücke und eignen sich daher weniger für einen Laien, da nicht anzunehmen ist, dass diesem die Prädikatenlogik bekannt sein dürfte.

4. Konzept

In der Wissensrepräsentation werden oft Ontologien verwendet, um Daten zu strukturieren und diese semantisch anzureichern. Zur Visualisierung dieser Ontologien wird ein Konzept benötigt, das sich nicht auf Teilaspekte einer Ontologie beschränkt, sondern Ontologien kompakt und ganzheitlich darstellen kann. Mit VOWL (Abschnitt 4.1) existiert bereits ein solches Konzept. Die derzeit im Entstehen befindliche Version 2.0 von VOWL (Abschnitt 4.2) wird im Rahmen dieser Arbeit als geeignetes Konzept für die kompakte und ganzheitliche Visualisierung von Ontologien angesehen. Sie dient daher als Konzept für die prototypische Umsetzung. Im Gegensatz zu den verwandten Arbeiten aus Abschnitt 3 stellt das hier vorgestellte VOWL Ontologie-Elemente in einer kompakteren Form dar. Zur Darstellung der Elemente werden wenige visuelle Elemente benötigt. Die verwendeten Elemente unterscheiden sich hinsichtlich Form und Farbgebung deutlich voneinander. Dadurch können diese durch den Benutzer leichter unterschieden und gleiche Elemente gruppiert werden. Dies soll das Lesen einer Ontologie erleichtern.
In den folgenden Abschnitten wird VOWL in Version 1.0 und 2.0 vorgestellt 2.0, sowie eine Optimierung von VOWL 1.0 vorgestellt, die im Rahmen dieser Arbeit entwickelt wurde.

4.1. VOWL 1.0

Die Visual Notation for OWL Ontologies (VOWL) ist eine Spezifikation zur Visualisierung von OWL Ontologien. Am 28. Januar 2013 wurde Version 1.0 veröffentlicht, an Version 2.0 wird derzeit gearbeitet.
VOWL 1.0 bietet ein Visualisierungskonzept, das in drei Ansichten strukturiert ist, die im Folgenden näher vorgestellt werden:

konzeptuellen Sicht: enthält Klassen, Eigenschaften und stellt die Beziehungen untereinander dar.

Instanzansicht: stellt Instanzen und ihre Beziehungen untereinander dar.

integrierte Ansicht: kombiniert beide Sichten und stellt sowohl die Klassen als auch ihre Instanzen samt Eigenschaften und deren Beziehungen untereinander dar.

Ein Beispiel der konzeptuellen Sicht ist in Abbildung 4.3 und ein Muster der integrierten Sicht in Abbildung 4.4 vorhanden.
In VOWL werden alle Elemente einer Ontologie grafisch repräsentiert. VOWL verwendet hierfür Knoten und Kanten. Knoten gibt es in runden und quadratischen Ausführungen. Runde Knoten stellen Klassen dar, während quadratische Knoten Literale darstellen und

Kanten Properties verkörpern. Meistens verbindet eine Kante zwei verschiedene Knoten und stellt damit eine Property der Ontologie dar. Abbildung 4.1 visualisiert die Darstellung von Klassen, Literale und Poperies in VOWL. Eine Kante kann im Falle einer `Subproperty` auch auf eine andere Kante zeigen. Falls Klassen Instanzen enthalten, so werden diese in der Instanzansicht und in der integrierten Ansicht durch Kreisausschnitte visuell repräsentiert, die Länge des Kreisbogens der jeweiligen Instanz richtet sich nach der Gesamtanzahl an Instanzen einer Klasse. Die einzelnen Instanzen stellen dabei einen Kreissektor dar. In allen Sichten wächst der Radius und damit die visuelle Größe einer Klasse falls diese Instanzen enthält. Dieses Verhalten wird in Abbildung 4.2 dargestellt.
Die visuelle Darstellung sämtlicher Elemente ist in der VOWL 1.0 Spezifikation enthalten [NL13]. VOWL nutzt das kräftebasierte Layout. Dadurch werden jene Bestandteile einer Ontologie, die wenig Verbindungen mit anderen Bestandteilen haben, am Rande der Darstellung platziert, während jene mit vielen Verbindungen im Zentrum dargestellt werden. Diese Form der Darstellung unterstützt den Benutzer im Verstehen einer Ontologie, da die wichtigsten Bestandteile einer Ontologie im Zentrum der Ansicht dargestellt werden.
VOWL 1.0 bietet eine integrierte Darstellung sowohl der Instanzen als auch des Konzepts einer Ontologie und bietet dem Benutzer so einen Überblick der Ontologie. `Object Properties` und `Datatype Properties` können in VOWL klar unterschieden werden. Die konzeptionellen Strukturen einer Ontologie sollen mit diesem Visualisierungskonzept klar ersichtlich werden [NL13].

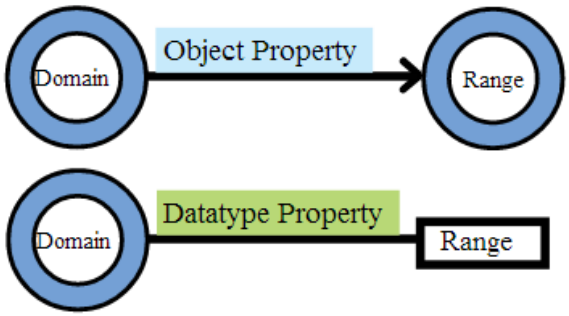


Abbildung 4.1.: Darstellung von Eigenschaften in VOWL 1.0 [NL13].

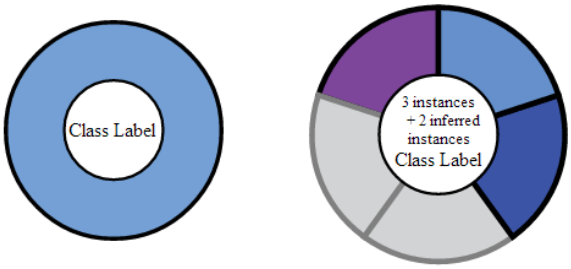


Abbildung 4.2.: Darstellung von Klassen (links) und Instanzen (rechts) in VOWL 1.0 [NL13].

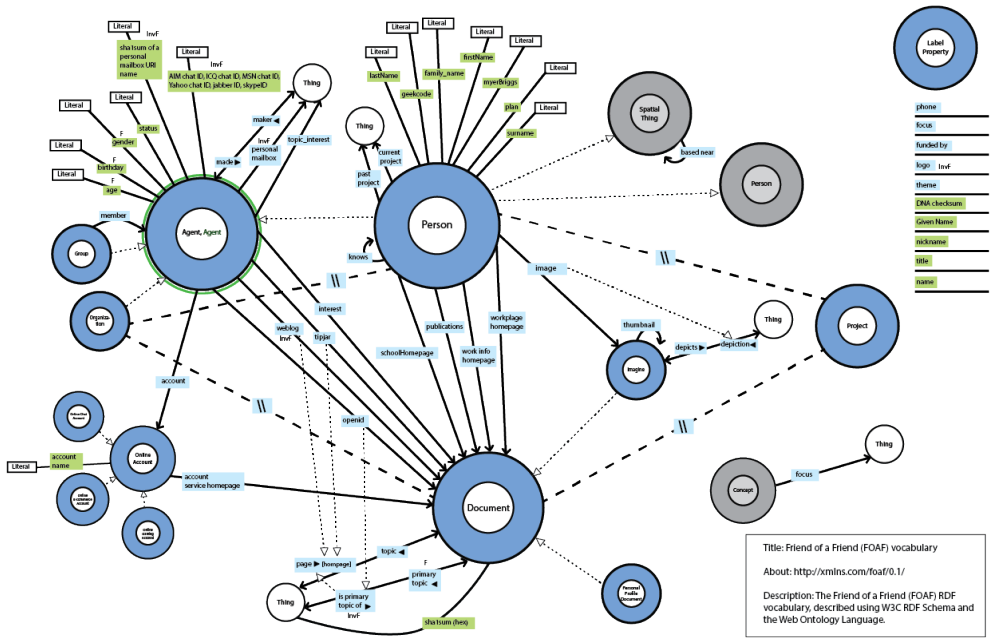


Abbildung 4.3.: Darstellung der konzeptuellen Sicht aus VOWL 1.0 [NL13].

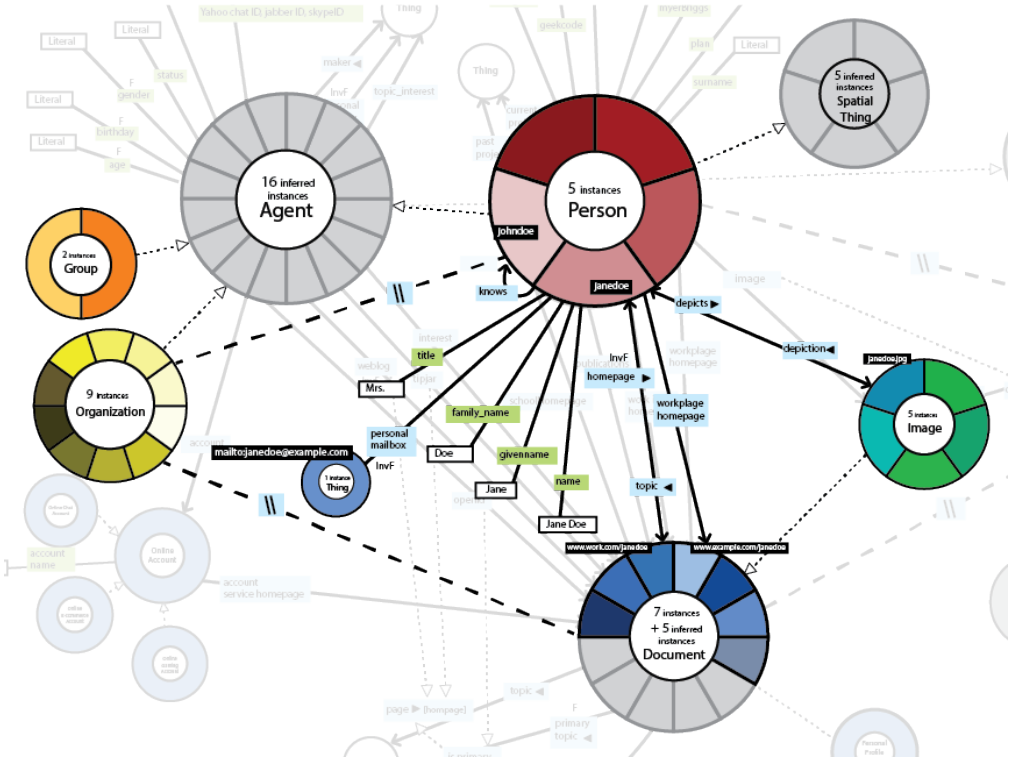


Abbildung 4.4.: Darstellung der integrierten Sicht aus VOWL 1.0 [NL13].

4.2. VOWL 2.0

VOWL 2.0 stellt eine Weiterentwicklung von VOWL 1.0 dar und enthält hauptsächlich eine vereinfachte und verständlichere Darstellung, auf die im Folgenden näher eingegangen wird. Nach derzeitigem Stand beinhaltet 2.0 beispielsweise keine integrierte Ansicht mehr. Enthält eine Klasse Instanzen, so werden diese nicht mehr als Kreissegment der Klasse dargestellt, stattdessen wird nur die Anzahl der Instanzen über die dargestellte Größe der Klasse visualisiert. Instanzen und ihre Instanzdaten werden erst nach Aufforderung des Nutzers dargestellt, dies kann beispielsweise über das Auswählen einer spezifischen Klasse geschehen. Allerdings wird darüber nachgedacht, die Darstellung der Instanzen als Kreissegmente optional weiterhin anzubieten.
Daneben vereinfacht VOWL 2.0 die visuelle Darstellung der grafischen Notation. So enthalten Klassen keinen inneren Ring mehr, sondern sind komplett gefüllt. Einen Ausschnitt der vereinfachten Darstellung wird in Abbildung 4.5 dargestellt. VOWL 2.0 befindet sich zurzeit noch in Entwicklung.

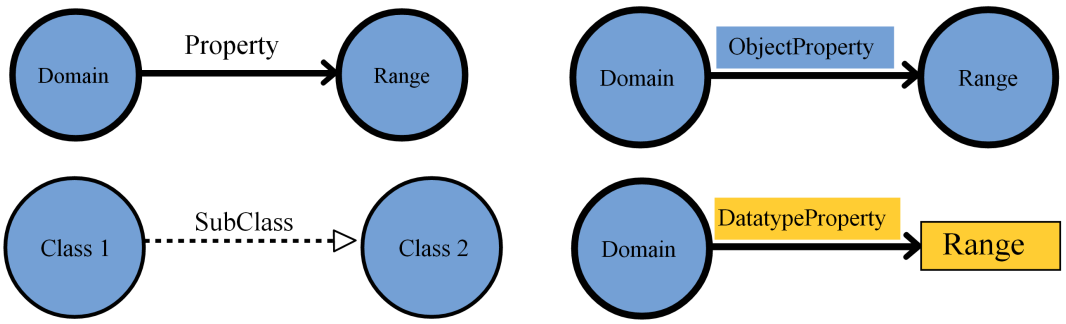


Abbildung 4.5.: Properties in VOWL 2.0.

4.3. Konzeptoptimierungen

In diesem Abschnitt wird das im Rahmen dieser Arbeit entwickelte Optimierungskonzept von VOWL vorgestellt und erläutert. Die Optimierungsvorschläge sind ebenfalls in die Weiterentwicklung von VOWL 1.0 eingeflossen und bilden eine von vielen Grundlagen für VOWL 2.0.
In der linken Grafik von Abbildung 4.6 wird die `subClassOf` Property von VOWL 1.0 gezeigt, während mittig die Optimierung abgebildet ist. Die rechte Darstellung aus Abbildung 4.6 zeigt die vorerst letzte Fassung dieser Property aus VOWL 2.0. Von Probanden der VOWL 1.0 Evaluation wurde ebenfalls vorgeschlagen, `subClassOf` Properties zusätzlich zu beschriften, da ihnen sonst die Bedeutung dieser Property unklar sei [NL13].
VOWL 1.0 regelt die Darstellung verschiedener symmetrischer Properties mit identischer Domain & Range nicht explizit. Die Konzeptoptimierung sieht die Darstellung dieser symmetrischen Properties verteilt um die Domain und Range Knoten vor. Alternativ hätten diese

auch, wie in Abbildung 4.7 symbolisiert, mit unterschiedlichem Abstand zum Knoten dargestellt werden können. Die Optimierung aus Abbildung 4.8 hat den Vorteil der leichteren Zuordnung der Kantenbeschriftung mit der Kante.
Weitere Anpassungen sehen den Austausch der Symbole für `owl:disjointWith`, `owl:unionOf`, `owl:intersectionOf` und `owl:complementOf` Properties vor. Anstelle jener Darstellung aus Abbildung 4.9 sieht die Optimierung die, der Mengenlehre entnommenen, Symbole aus Abbildung 4.10 vor. In Abbildung 4.9 ist links oben `owl:disjointWith`, links unten `owl:unionOf`, rechts oben `owl:intersectionOf` und rechts unten `owl:complementOf` dargestellt. In Abbildung 4.9 ist links oben `owl:unionOf`, links unten `owl:intersectionOf`, rechts oben `owl:disjointWith` und rechts unten `owl:complementOf` dargestellt. Mit Ausnahme von `owl:complementOf` sind alle Symbole der Mengenlehre entnommen während `owl:complementOf` durch invertierte Symbole und Farben eine Invertierung und damit ein Komplement verkörpern soll.
Des Weiteren sieht ein Entwurf die Darstellung verschiedener Properties mit denselben Domain- und Range-Angaben gemäß Abbildung 4.11 vor. Im Gegensatz zu den Darstellungen dieser Properties aus Abbildung 4.2 und Abbildung 4.3 sollen diese als Bündel visualisiert werden.
Die Größe einer Klasse spiegelt die Anzahl der darin enthaltenen Instanzen wieder. Zur Darstellung der eigentlichen Instanzdaten wird die Verwendung eines seitlichen angebrachten Informationsfensters vorgesehen. Der Benutzer soll auf diese Weise einen ganzheitlichen Überblick der Ontologie erhalten, samt Übersicht über die Anzahl der Instanzen. Die Details der eigentlichen Instanzen können explorativ erforscht werden, sie sollen den Benutzer nicht durch eine mögliche Informationsüberflutung am Verstehen des Konzeptes der Ontologie hindern. Dieser Aufbau wird in Abbildung 4.12 skizziert.

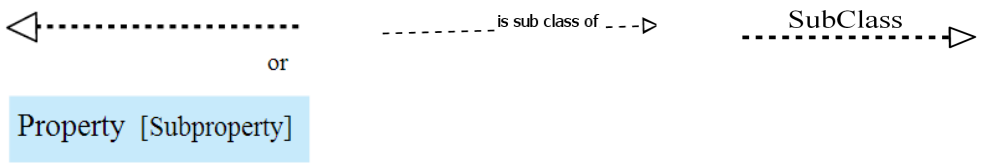


Abbildung 4.6.: Unterschiedliche Versionen für `subPropertyOf`.

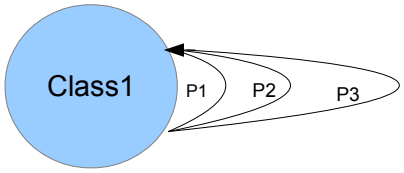


Abbildung 4.7.: Alternative Darstellung mehrfacher symmetrischer Properties.

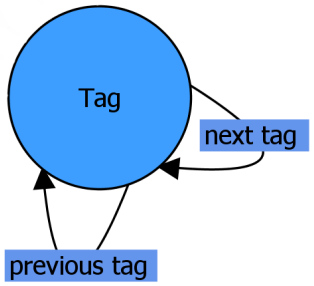


Abbildung 4.8.: Darstellung mehrfacher symmetrischer Properties.

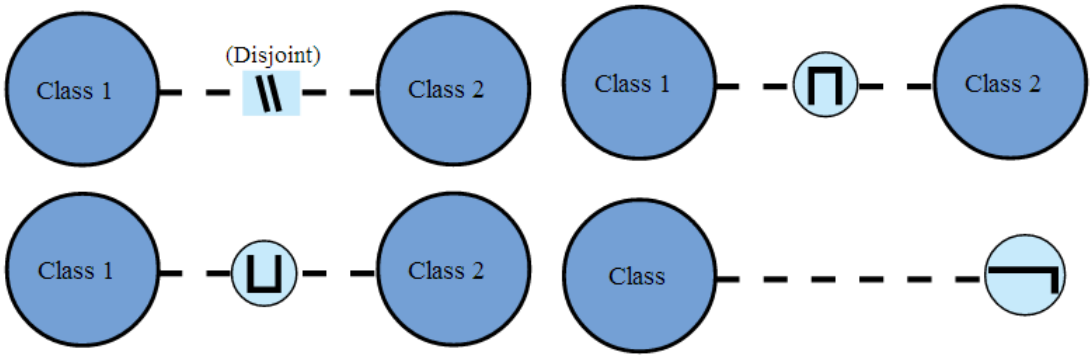


Abbildung 4.9.: Darstellung von `owl:disjointWith` (links oben), `owl:unionOf` (links unten), `owl:intersectionOf` (rechts oben) und `owl:complementOf` (rechts unten) in OWL 1.0.

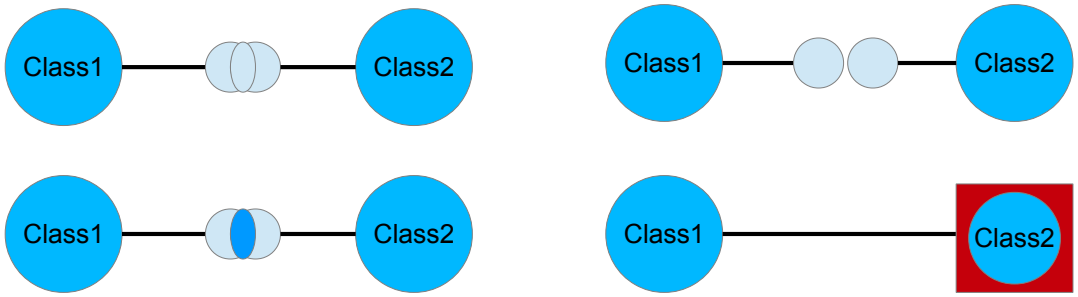


Abbildung 4.10.: Darstellung von `owl:unionOf` (links oben), `owl:intersectionOf` (links unten), `owl:disjointWith` (rechts oben), und `owl:complementOf` (rechts unten) nach Optimierung des Visualisierungskonzeptes.

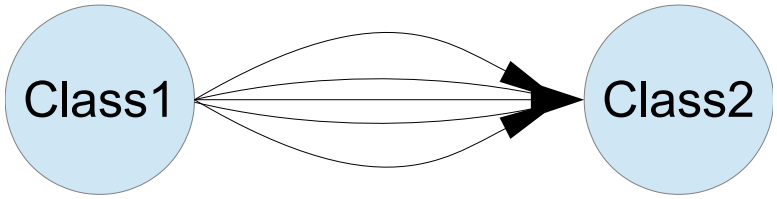


Abbildung 4.11.: Darstellung mehrfacher Properties gemäß Konzeptoptimierung.

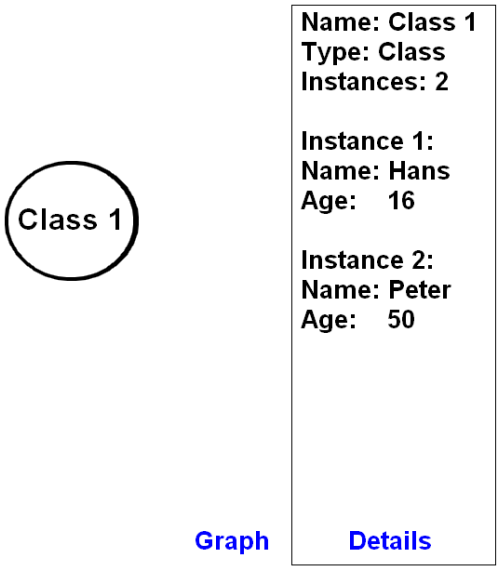


Abbildung 4.12.: Skizze der explorierbaren Detailansicht.

5. Frameworks für Graphen

Im Rahmen dieser Arbeit müssen Ontologien visualisiert werden. Die Visualisierung soll durch die Verwendung von Grafikframeworks erleichtert werden, schließlich müssen Grundoperationen nicht selbst implementiert werden. Grafikframeworks bieten unter anderem Funktionen, um eine Kante zwischen zwei Knoten zeichnen zu lassen. Falls man eine derartige Funktion selbst implementieren möchte, so wird beispielsweise die Kenntnis der genauen Position des Knoten notwendig. Daneben implementieren Grafikframeworks unzählige weitere Funktionen, welche die Entwicklung einer Ontologievisualisierung unterstützen können.
Aus diesem Anlass werden verschiedene Frameworks auf ihre Eignung, die Implementierung dieser Visualisierung zu unterstützen, untersucht. Hinsichtlich der Bewertung der Frameworks werden folgende Kriterien aufgestellt:

Einarbeitungszeit: Die Einarbeitung in die Grundlagen des entsprechenden Frameworks sollte aufgrund des zeitlich beschränkten Rahmens dieser Arbeit möglichst wenig Zeit beanspruchen. Mit dem jeweiligen Grafikframework sollte innerhalb von drei Stunden ein Prototyp zur Visualisierung eines einfachen Graphen realisiert werden. Die Verwendung des Frameworks sollte in einer positiven Kosten/Nutzen Rechnung resultieren, d.h.: die Verwendung des Frameworks sollte zeitsparender als die direkte Verwendung von Java2D samt Implementierung aller benötigten Features sein.

Dokumentation: Das Framework sollte über eine umfangreiche Dokumentation verfügen. Hierbei wird nicht nur die Dokumentation des Frameworkquellcodes gewertet, sondern auch alle Dokumente über dessen Verwendung, seien es Beispiele, Grundlagen oder theoretische Modelle. Der Ursprung der Dokumentation, ob vom Hersteller oder von Dritten, spielt dabei keine Rolle.

Editierbarkeit (Graph): Das Framework sollte ausreichend viele Möglichkeiten bieten, die Visualisierung des Graphen zu beeinflussen. Beispielsweise sollte das Framework verschiedene Parameter für die Schriftfarbe, Schriftgröße oder Liniendicke akzeptieren. Wie die Beeinflussung der zu erstellenden Visualisierung im Detail erfolgt, ist irrelevant.

Editierbarkeit (Code): Wenn eine erwünschte Modifikation der Visualisierung nicht über das Framework selbst realisiert werden kann, so wird eine Ergänzung des Codes des Frameworks notwendig. Diese Veränderungen sollten ohne größere Schwierigkeiten möglich sein. Neben der Erreichbarkeit des Quellcodes zählt zu diesem Kriterium auch, wie umfangreich die Einarbeitung in das Framework ist. Frameworks, deren Module gut gekapselt und deren Funktionalität klar erkennbar sind, sind in der Regel leichter zu verstehen und damit zu verändern als Module, deren Zweck völlig unklar sind.

Performance: Die Erstellung des Graphen durch das Framework sollte so flott geschehen, dass der Nutzer von der Generierung des Graphen nichts mitbekommt. Handelt es sich bei dem Framework um ein interaktives Framework, so sollte die Interaktion mit dem Graphen zu keiner, vom Benutzer zu bemerkenden, Verzögerung führen.

Interaktivität: Der Nutzer sollte in der Lage sein mit dem Graphen zu interagieren. Die Interaktion sollte den Nutzer bei der Exploration der Ontologie unterstützen. Das Framework muss dem Benutzer beispielsweise die Möglichkeit bieten, Bestandteile des Graphen visuell zu verschieben. Die Interaktivität ist auf das Explorieren beschränkt. Eine tiefer gehende Modifikation des Graphen ist nicht vorgesehen, der Nutzer muss keine Elemente aus dem Graphen löschen oder hinzufügen können.

Integrierbarkeit: Das Framework zur Generierung des Graphen sollte in die eigene Anwendung integrierbar sein.

Innerhalb dieses Kapitels werden Frameworks hinsichtlich der hier aufgestellten Kriterien bewertet. Im Anschluss an die Vorstellung und Bewertung der einzelnen Frameworks folgt eine Zusammenfassung der untersuchten Frameworks, bezüglich ihrer Eignung die Visualisierung von Ontologien im Rahmen dieser Arbeit sinnvoll zu unterstützen.

5.1. GraphViz

GraphViz ist eine Sammlung von Programmen zur Visualisierung von Graphen. Ihre Entwicklung begann 1988 bei AT&T und den Bell-Labs. Lizenziert wird GraphViz unter der Eclipse Public License[1]. Die Programmsammlung enthält verschiedene Teilprogramme, die jeweils eine andere Art von Graphen generieren [Gra13a]. Im Folgenden werden die verschiedenen Programmsammlungen und die Graphen, die sie erzeugen können, vorgestellt:

dot: erstellt hierarchische Strukturen. Dot versucht bei der Graphengenerierung einen Graphen zu erstellen, dessen Kanten möglichst kurz sind, sich nicht gegenseitig schneiden und in dieselbe Richtung zeigen - von oben nach unten oder von links nach rechts.

neato: generiert kräftebasierte Layouts. Neato sollte nach [Gra13a] nicht für Graphen mit mehr als 100 Knoten verwendet werden. Bei der Graphengenerierung wird eine multidimensionale Skalierung eingesetzt. Dabei werden ähnliche Objekte räumlich näher, unähnliche eher entfernt voneinander platziert. Nach diesem Ansatz werden verbundene Knoten innerhalb eines Graphen räumlich näher und unverbundene Knoten räumlich eher getrennt voneinander angeordnet.

fdp: erstellt hierarchische Strukturen. Im Gegensatz zu neato implementiert fdp eine Fruchterman-Reingold Heuristik und ist damit auch für größere Graphen geeignet. Bei diesem Ansatz werden Anziehungskräfte und Abstoßungskräfte zwischen Knoten eines Graphen berechnet und visualisert.

[1] http://www.graphviz.org/Download.php

twopi: erstellt einen Graphen mit einem radialen Layout.

circo: erstellt einen Graphen mit einem circulären Layout.

Jedes Teilprogramm der GraphViz-Programmsammlung generiert einen Graphen aus einer textuellen Beschreibung [Nor04]. Diese Beschreibung erfolgt in der Auszeichnungssprache DOT. Nach erfolgreicher Generierung wird das Ergebnis als Datei exportiert [KN+91]. Beispielsweise enthält Quelltext 5.1 einen in der Auszeichnungssprache DOT beschriebenen Graph. DOT generiert daraus einen Graph, aus dem ein visueller Graph, wie in Abbildung 5.1 dargestellt, erstellt wird.

Quelltext 5.1 Ein Beispiel eines in DOT beschriebenen Graphen, entnommen aus [KN+91].

```
digraph G {
main -> parse -> execute;
main -> init;
main -> cleanup;
execute -> make_string;
execute -> printf;
init -> make_string;
main -> printf;
execute -> compare;
: }
```

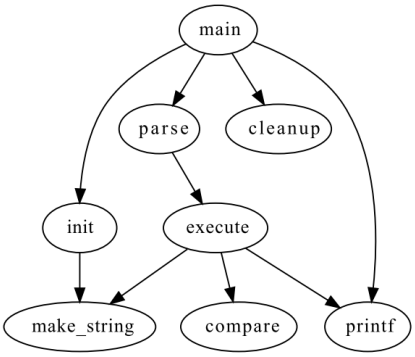


Abbildung 5.1.: Der in Quelltext 5.1 beschriebene Graph nach der Generierung durch dot, Abbildung entnommen aus [KN+91].

GraphViz wird stetig weiter entwickelt, so erschien am 1. August 2013 GraphViz in Version 2.32 [Gra13a]. GraphViz dient bereits einigen Protégé Plug-ins als Framework zur Visualisierung von Graphen. Beispielsweise benötigt das schon in Abbildung 2.2 aus Abschnitt 2.1 erwähnte Plug-in OWLViz GraphViz [Hor10].
Die Dokumentation zu GraphViz ist ausführlich und umfangreich. Die Webseite enthält nicht nur eine umfangreiche Galerie bereits generierter Graphen, sondern auch detaillierte Handbücher zu den unterschiedlichen Bestandteilen von GraphViz und eine detaillierte Beschreibung der verschiedenen Parameter. Die einfache und beschreibende Struktur der

Beschreibungssprache DOT, sowie die detaillierte Dokumentation von GraphViz ermöglichen einen einfachen Einstieg in die Erstellung unterschiedlicher, nach eigenen Wünschen angepasster, Graphen. [Gra13b] enthält beispielsweise eine umfangreiche Liste verschiedener Formen von Knoten und Kanten, die mit GraphViz visualisiert werden können. Als Open-Source-Anwendung liegt GraphViz zwar im Quellcode vor[2], jedoch sind Änderungen am Code mit einer erneuten Kompilierung verbunden, wofür wiederum zahlreiche Pakete und Werkzeuge benötigt werden. Des Weiteren werden Kenntnisse in der Programmiersprache C vorausgesetzt [Gan13]. Die Performance von GraphViz ist mehr als ausreichend, beispielsweise wurde der Graph in Abbildung 5.2 durch neato innerhalb von 0.41 Sekunden generiert [Nor04]. Da GraphViz als eine in C implementierte Anwendung vorliegt, ist sowohl die Interaktivität als auch die Integrierbarkeit zwischen der C-Anwendung und der Java-Anwendung eingeschränkt.

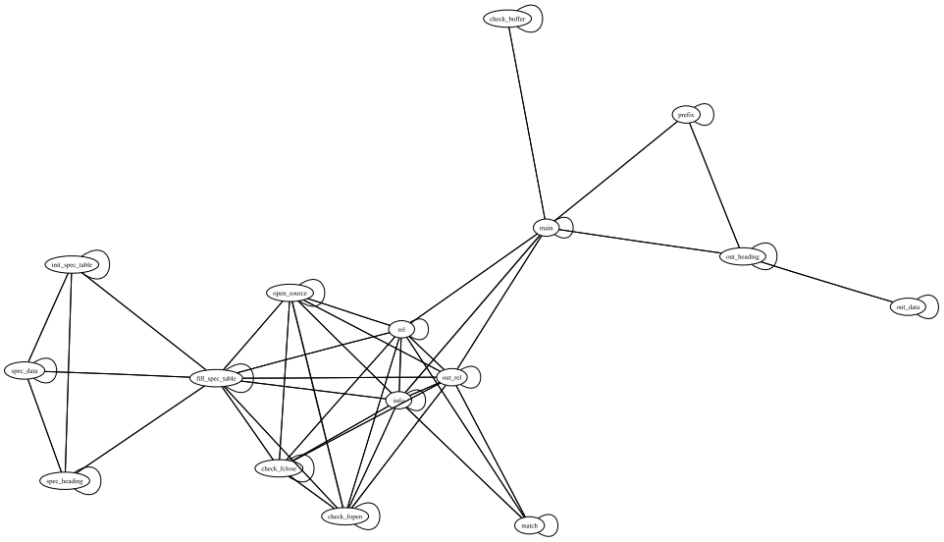


Abbildung 5.2.: Ein innerhalb 0,41 Sekunden generierter Graph, Abbildung entnommen aus [Nor04].

Für GraphViz wurden einige Java-Interfaces entwickelt, mit deren Hilfe die GraphViz-Programmsammlung direkt in Java verwendet werden kann. Beispiele für derartige Interfaces sind:

Grappa: GRAPh PAckage in Java [BML97]

JPGD: Java-basierter Parser für GraphViz Dokumente[3]

GraphViz Java-Implementierung[4]

[2] http://www.graphviz.org/Download_source.php
[3] http://www.alexander-merz.com/graphviz

Die Dokumentationen zu diesen Interfaces sind dürftig. Für Gappa konnte beispielsweise neben dem Quellcode nur ein einziges Paper gefunden werden. Es gibt nahezu keine Informationen über den Umfang mit dem JPGD bzw. die GraphViz Java-Implementierung die Auszeichnungssprache DOT verarbeiten kann.

5.2. Grappa

Grappa ist ein Java-Package mit den Graphen, die in der Auszeichnungssprache DOT beschrieben wurden, visualisiert werden können. Ebenso wie GraphViz wurde Grappa von AT&T entwickelt. Die Webseite enthält neben dem Quellcode auch eine Online-Demonstration, mit der Grappa getestet werden kann.
Laut Webseite existiert zu Grappa neben dem Paper [BML97] und JavaDocs[5] keine weitere Dokumentation. Die Webseite weist ebenfalls darauf hin, dass Grappa kein Layout beherrscht[6]. Abbildung 5.3 zeigt den Aufbau, während Abbildung 5.4 die Klassenhierarchie von Grappa beschreibt. Grappa wird von dem Protégé-Visualisierungsplugin OntoViz verwendet.
Die letzte Aktualisierung von Grappa fand am 22. September 2010 statt, 2008 gab es kleinere Verbesserungen, die aktive Entwicklung schien zwischen 2001 und 2006 stattgefunden zu haben.
Innerhalb der gesetzten Frist für die Einarbeitungszeit konnte kein Graph generiert werden. Die Einarbeitung scheiterte unter anderem an fehlenden Paketen, die zusammengesucht werden mussten. Beispielsweise benötigt Grappa das Paket java_cup.runtime, welches ebenfalls von AT&T entwickelt wurde[7]. Innerhalb der gesetzten Frist konnten nicht alle benötigten Pakete gefunden werden.
Aufgrund der Verwendung der Auszeichnungssprache DOT ist davon auszugehen, dass die mit Grappa zu visualisierende Graphen hinreichend an eigene Wünsche angepasst werden können. Aufgrund der teils widersprüchlichen Dokumentation fehlen jedoch klare Aussagen, ob alle Deklarationen aus DOT unterstützt werden. Die Webseite deutet beispielsweise an, dass eine große Menge unterschiedlicher Knotenformen unterstützt werden. Soll diese Information den Leser darüber informieren, dass Grappa verglichen mit anderen Frameworks viele Formen unterstützt, oder soll die Information darauf hinweisen, dass eben nicht alle Formen verwendet werden können?
Aufgrund der, in Abbildung 5.3 erwähnten Client-Server Trennung erschien eine eventuell benötigte Modifikation des Codes als zu zeitaufwendig und hätte möglicherweise den zeitlichen Rahmen dieser Arbeit gesprengt. Aufgrund der Verwendung von Grappa in OntoGraf kann die Performance von Grappa allerdings als ausreichend bewertet werden. Die Onlinedemonstration beweist die Existenz von Tooltips, allerdings scheint der Benutzer keine Knoten verschieben zu können. Die Online-Demonstration erscheint recht rudimentär, das Kompilieren des Graphen aus Quelltext 5.1 ist nicht möglich, obwohl dieser Graph der GraphViz-Dokumentation entnommen wurde [KN+91]. Nicht abschließend geklärt werden

[5] http://www2.research.att.com/~john/Grappa/docs/att/grappa/package-summary.html
[6] http://www2.research.att.com/~john/Grappa/grappa_faq.html#Usage_Q1
[7] http://www2.research.att.com/~john/Grappa/docs/java_cup/runtime/package-summary.html

konnte, ob dieser Fehler auf eine nicht ausreichende Unterstürzung der Auszeichnungssprache DOT hindeutet oder ob ein Fehler innerhalb der Online-Demonstration vorliegt.

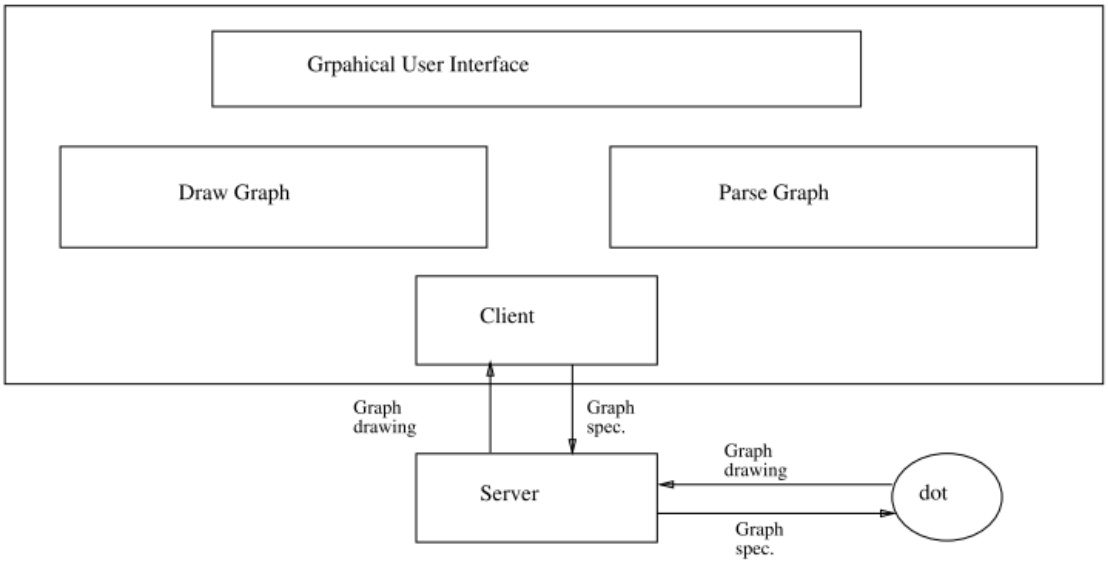


Abbildung 5.3.: Aufbau von Grappa, entnommen aus [BML97].

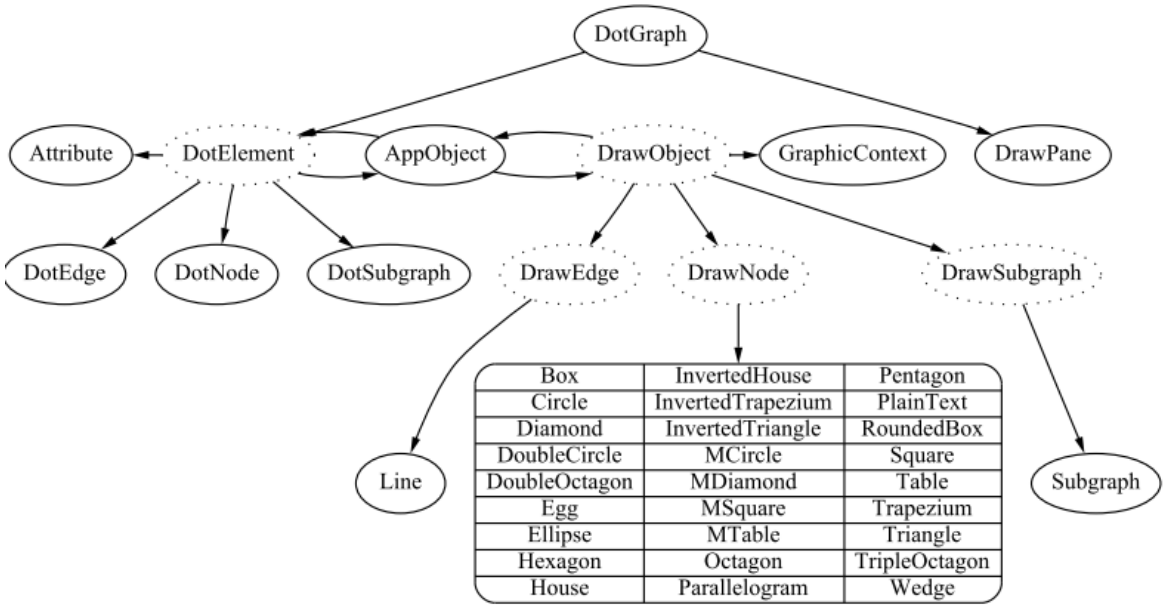


Abbildung 5.4.: Klassenhierachie in Grappe, entnommen aus [BML97].

5.3. Eclipse Zest

Bei Zest, dem Eclipse Visualization Toolkit [Ste13], handelt es sich um eine Sammlung von Visualisierungskomponenten für Eclipse. Wie Quelltext 5.2 zeigt, unterstützt Zest Graphen, die in der Auszeichnungssprache DOT beschrieben werden können. Eine objektorientierte Erstellung der Komponenten ist, wie in Quelltext 5.3 gezeigt, ebenfalls möglich.

Quelltext 5.2 Ein Beispiel für die Verwendung von DOT in Zest, Beispiel [Ste13] entnommen.

```
Shell shell = new Shell();
DotGraph graph = new DotGraph("digraph{ 1->2 }", shell, SWT.NONE);
graph.add("2->3").add("2->4");
graph.add("node[label=zested]; edge[style=dashed]; 3->5; 4->6");
open(shell);
```

Quelltext 5.3 Objektorientierte Weise einen Graphen zu Erstellen, Beispiel [Vog11] entnommen.

```
graph = new Graph(parent, SWT.NONE);
GraphNode node1 = new GraphNode(graph, SWT.NONE, "A");
GraphNode node2 = new GraphNode(graph, SWT.NONE, "B");
GraphNode node3 = new GraphNode(graph, SWT.NONE, "C");
new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node1, node2);
new GraphConnection(graph, SWT.NONE, node3, node1);
```

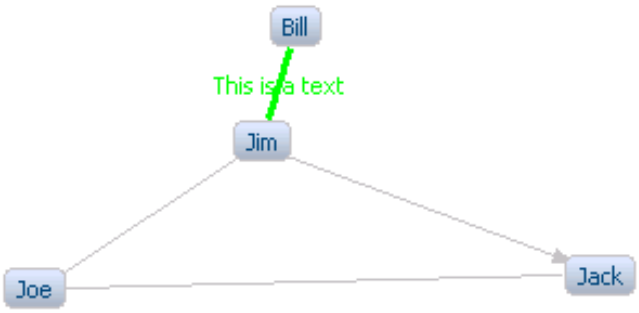


Abbildung 5.5.: Zest visualisiert einen Graphen [Vog11].

Die verfügbaren Informationen zu Zest sind deutlich geringer als jene zu GraphViz. Die verfügbaren Dokumente zu Zest sind größtenteils Anleitungen, beschreiben, jedoch nicht ins Detail gehen. Einige der auf der Zest-Webseite verlinkten Dokumente sind nicht mehr verfügbar. Innerhalb der Zeitvorgabe konnte kein Graph mittels Zest generiert werden, sodass die Einarbeitungszeit als negativ bewertet werden muss. Graphen lassen sich in Zest nur eingeschränkt editieren. Zest unterstützt die Auszeichnungssprache DOT, allerdings gibt es keine Informationen, ob alle Bestandteile von DOT unterstützt werden oder nur eine Teilmenge. Informationen über die objektorientierte Graphenerstellung sind ebenfalls nicht verfügbar.

5.4. JGraph

JGraph ist ein auf Java basiertes Grafikframework zur Visualisierung von Graphen, zu dem verschiedenen Versionen existieren. 2002 erschien die erste Version von JGraph. Ab Version 5 wird die Entwicklung unter dem Namen mxGraph kommerziell fortgesetzt, während die Open-Source-Variante ab Version 6 den Namen JGraphX trägt. Innerhalb dieser Ausarbeitung werden alle Versionen unter dem Namen JGraph geführt, da eine detaillierte Unterscheidung der einzelnen Varianten für die weitere Vorstellung und Analyse im Rahmen dieser Arbeit nicht relevant ist.
Quelltext 5.4 zeigt den Code eines, in der Einarbeitungszeit erstellten rudimentären Graphen dessen Visualisierung in Abbildung 5.6 vorgestellt wird. Die Dokumentation zu JGraph ist umfangreich, [Ald03] erklärt beispielsweise, wie man in JGraph einen Graphen erstellt, wie dieser an die eigenen Wünsche angepasst werden kann und wie JGraph auf Eingaben des Benutzers reagieren soll. [Ald02] beschreibt den Aufbau von JGraph näher. Weitere Dokumente sind im World Wide Web verfügbar. JGraph ist gut in die eigene Anwendung integrierbar, es werden nur wenige Pakete vorausgesetzt. Die Interaktivität von JGraph ist schwierig zu bewerten. Wie in Abbildung 5.7 gezeigt, kann der Benutzer die Größe von einzelnen Knoten verändern, das Verschieben der Knoten ist allerdings nicht möglich. In den Standardeinstellungen kann der Benutzer, wie in Abbildung 5.8 gezeigt, weitere Kanten hinzufügen, die beliebig im Raum verteilt werden können. Der Namen eines Knoten kann ebenfalls vom Nutzer verändert werden. Dieses Verhalten von JGraph ist aber abschaltbar. Letztendlich überwiegt der Eindruck, dass vieles in JGraph möglich ist, deren Implementierung jedoch recht umständlich ist, beispielsweise sind Tooltips, verglichen mit anderen Frameworks eher umständlich zu realisieren [Ald03, S. 10]. Die Performance von JGraph ist mehr als ausreichend.

Quelltext 5.4 Erstellung eines Demonstrationsgraphen mittels JGraph.

```
    mxGraph graph = new mxGraph();
    Object parent = graph.getDefaultParent();
    graph.getModel().beginUpdate();
    try {
           // graph.insertEdge(parent, id, value, source, target)
           Object v1 = graph.insertVertex(parent, null, "Node1", 20, 20, 80, 30);
           Object v2 = graph.insertVertex(parent, null, "Node2", 240, 150, 80, 30);
           Object v3 = graph.insertVertex(parent, null, "Node3", 170, 100, 80, 30);
           graph.insertEdge(parent, null, "V12", v1, v2);
           graph.insertEdge(parent, null, "V13", v1, v3);
     } finally {
           graph.getModel().endUpdate();
     }
     mxGraphComponent graphComponent = new mxGraphComponent(graph);
     JGraphX frame = new JGraphX();
frame.add(graphComponent);
frame.pack();
frame.setVisible(true);
```

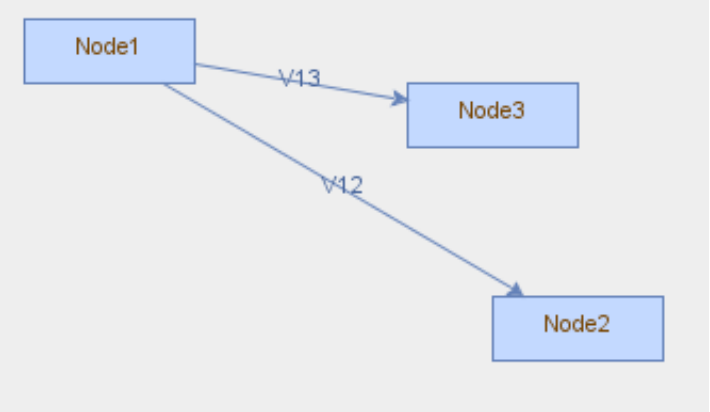


Abbildung 5.6.: Visualisierung des Codes aus Quelltext 5.4 mittels JGraph.

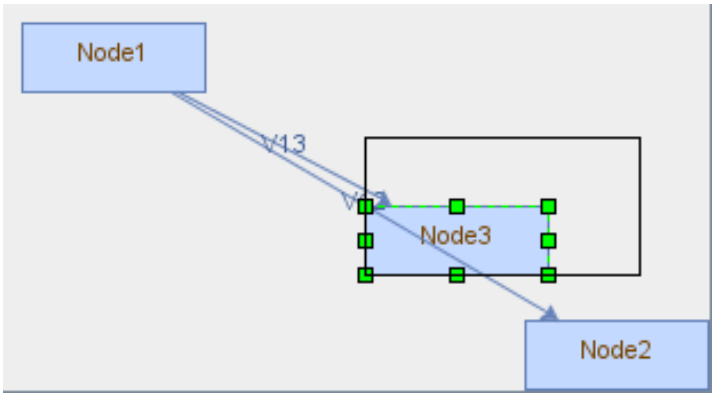


Abbildung 5.7.: Visualisierung des Codes aus Quelltext 5.4 nach Veränderung der Größe des dritten Knoten.

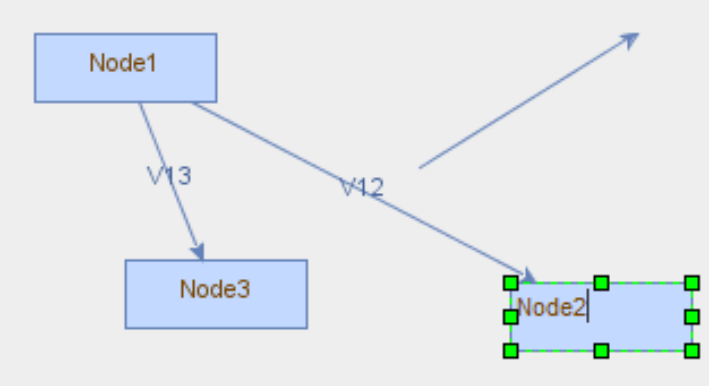


Abbildung 5.8.: Freie Kanten im Raum und Namensänderung.

5.5. JUNG

Java Universal Network/Graph Framework (JUNG) ist ein auf Java basiertes Framework zur Visualisierung von Graphen und Netzwerken. JUNG findet beispielsweise Anwendung in der Analyse von Netzwerkdaten [Jun]. Innerhalb der Einarbeitungszeit konnte mittels JUNG der Graph aus Abbildung 5.9 auf Basis des Codes aus Quelltext 5.5 generiert werden. Komplexere Graphen lassen sich ebenfalls mit JUNG visualisieren. Abbildung 5.10 veranschaulicht hierzu ein Beispiel eines komplexeren Graphen, der mittels JUNG visualisiert wurde.
Die JUNG-Dokumentation erscheint für das Framework angemessen umfangreich, ist jedoch deutlich geringer als die der beiden bereits vorgestellten Frameworks GraphViz und JGraph. Die JUNG-Webseite [JUN10b] hält einige Anleitungen parat. JUNG ist das einzige untersuchte Framework, dessen Anleitung von einem Entwickler des Frameworks geschrieben wurde [Ber10]. Das JUNG-Framework verfügt über eine aktivere Community. Bei Stackoverflow[8] wurden beispielsweise 260 Fragen mit dem Tag „JUNG“ versehen. Im Vergleich hierzu verfügt GraphViz mit mehr als 803 Tags über deutlich mehr Fragen auf Stackoverflow. Die Frameworks JGraph, Prefuse und Zest bilden mit 30, 61 und 41 Tags auf Stackoverflow deutlich das Schlusslicht. Auf Basis der auf Stackoverflow gestellten Fragen und Antworten lässt sich auf die Größe und Aktivität der Community der entsprechende Rückschluss ziehen, dass JUNG von den hier vorgestellten Frameworks die aktivste Community besitzt. JUNG dient nicht nur einer Vielzahl von Anwendungen als Grafikframework, sondern ist auch Bestandteil zahlreicher wissenschaftlicher Abhandlungen [Jun10a].
Der Graph scheint durch zahlreiche Parameter gut konfigurierbar zu sein. Dasselbe trifft auch auf die Editierbarkeit des Codes zu. Beispiele zu beiden Fällen können in [Ber10] gefunden werden. Die Performance von JUNG kann als ausreichend bewertet werden. Die Integrierbarkeit in die eigene Anwendung stellt kein Problem dar, auch wenn hierzu 17 JAR-Pakete importiert werden müssen. Die Pakete sind leicht aufzufinden und müssen nicht extra identifiziert und gesucht werden. JUNG verfügt ebenfalls über umfangreiche Möglichkeiten für Nutzer zur Interaktion mit dem visualisierten Graphen.

[8] http://stackoverflow.com

Quelltext 5.5 Erstellung eines Demonstrationsgraphen mittels JUNG.

```
private static Graph<String, Integer> getGraph() {
        final Graph<String, Integer> g = new DirectedSparseGraph<String, Integer>();
        g.addVertex("Haus");
        g.addVertex("Reihenhaus");
        g.addVertex("Mehrfamilienhaus");
        g.addVertex("Hochhaus");
        g.addVertex("kaputtes Haus");

        g.addEdge(1, "Haus", "Hochhaus");
        g.addEdge(2, "Reihenhaus", "Mehrfamilienhaus");
        g.addEdge(3, "kaputtes Haus", "Haus");

        return g;
}

private static GraphZoomScrollPane getJungGraphPane() {
        final Graph<String, Integer> g = getGraph();
        final VisualizationViewer<String, Integer> viewer = new
            VisualizationViewer<String, Integer>(
                        new CircleLayout<String, Integer>(g));
        viewer.setDoubleBuffered(true);
        final GraphZoomScrollPane paneWithGraph = new GraphZoomScrollPane(
                        viewer);
        return paneWithGraph;
}

@Override
protected Control createContents(final Composite parent) {
        final Composite graphViewComposite = new Composite(parent, SWT.NONE
                        | SWT.EMBEDDED);
        final Frame graphFrame = SWT_AWT.new_Frame(graphViewComposite);

        GridDataFactory.fillDefaults().grab(true, true)
                        .applyTo(graphViewComposite);

        graphFrame.add(getJungGraphPane());
        graphFrame.pack();
        graphFrame.setVisible(true);
        return parent;
}

public void run() {
        this.setBlockOnOpen(true);
        this.open();
        Display.getCurrent().dispose();
}

public static void main(final String[] args) {
        new Jung().run();
}
```

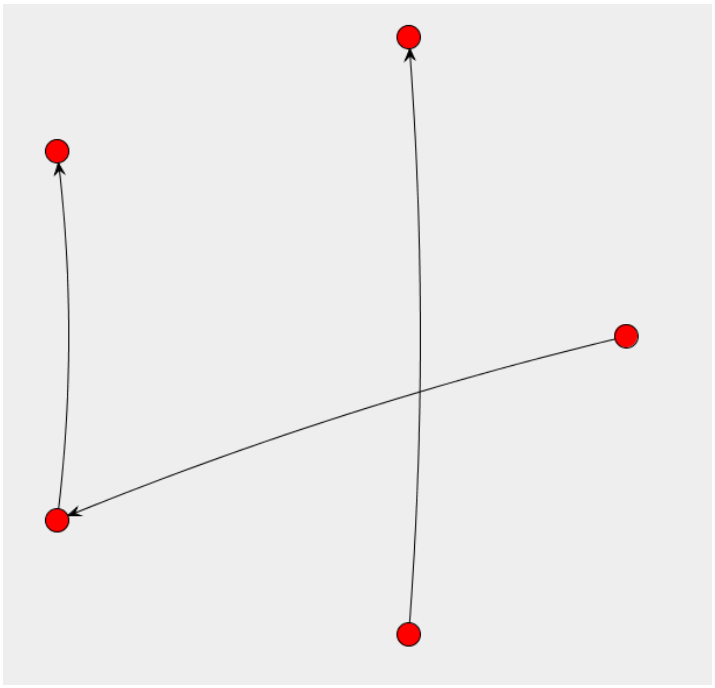


Abbildung 5.9.: Visualisierung des Codes aus Quelltext 5.5 mittels JUNG.

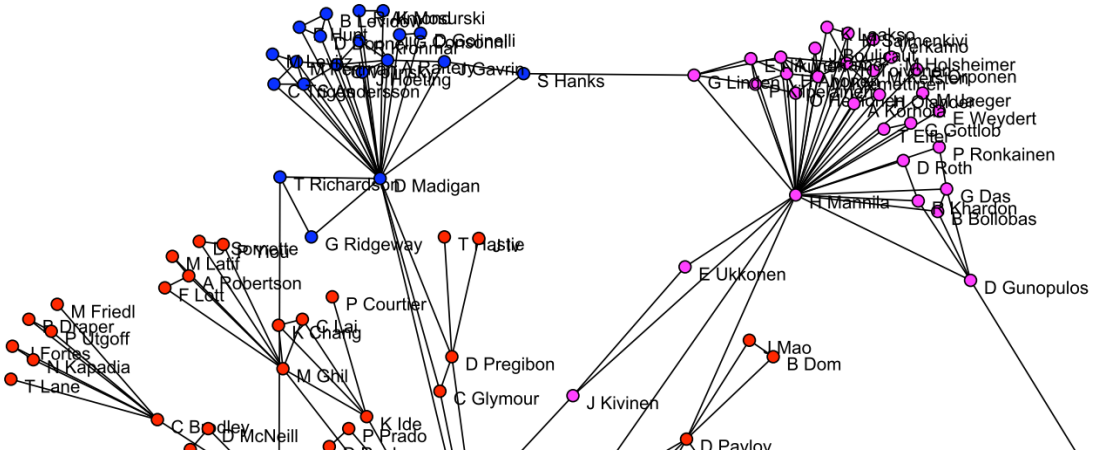


Abbildung 5.10.: Visualisierung eines komplexeren Graphen mittels JUNG, entnommen aus [Jun].

5.6. Prefuse

Prefuse ist ein auf Java basiertes Grafikframework, das unter BSD Lizenz[9] steht und daher sowohl nicht-kommerziell als auch kommerziell verwendet werden kann. Prefuse nutzt zur Visualisierung die Java 2D-Grafikbibliothek. Entwickelt wurde das Prefuse-Framework zur dynamischen Visualisierung von strukturierten und unstrukturierten Daten [HCL05]. Die Webseite des Frameworks hält neben einem Handbuch auch eine Galery verschiedener Visualisierungen parat [Ber13].
Der Quellcode demonstriert durch zahlreiche Beispiele verschiedener Visualisierungen die Mächtigkeit des Frameworks. Als Beispiel hierfür können die enthaltene Visualisierung der geografischen Verteilung von Postleitzahlen in den Vereinigten Staaten und eine Visualisierung einer Treemap gelten. Abbildung 5.11 und Abbildung 5.12 demonstrieren die Fähigkeiten von Prefuse zur Visualisierung von Graphen. Auch diese Beispiele sind im Quellcode des Prefuse-Frameworks vorzufinden, ihre Abbildungen können aus diesem generiert werden [Ber13].

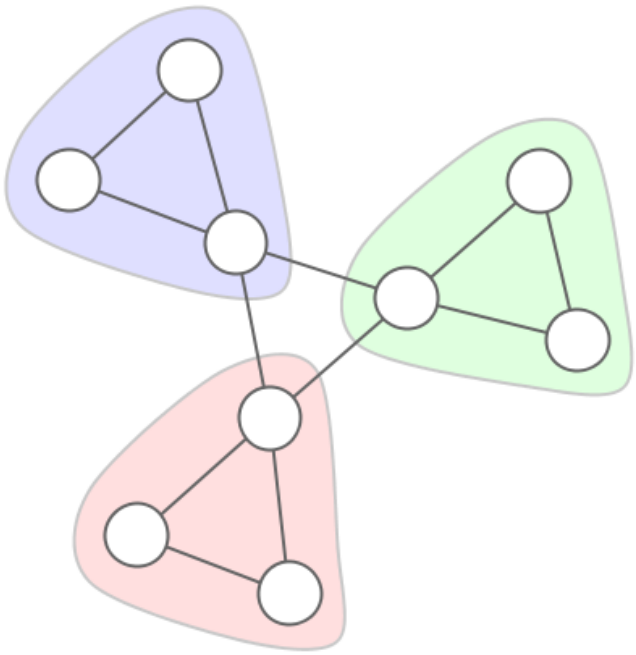


Abbildung 5.11.: Mittels Prefuse erstellte Visualisierung eines aggregierten Graphen.

[9] http://opensource.org/licenses/bsd-license.php

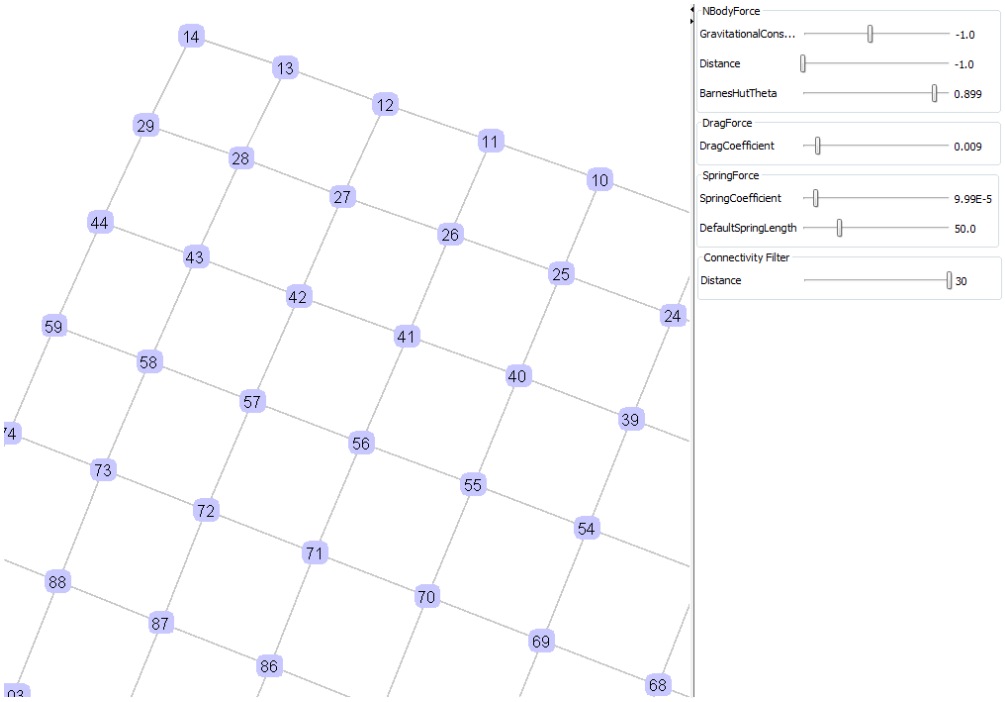


Abbildung 5.12.: Mittels Prefuse erstellte Visualisierung eines Graphen.

Innerhalb der festgelegten Frist zur Einarbeitung konnte der Codes aus Quelltext 5.6 und damit der Graph aus Abbildung 5.13 erstellt werden. In diesem Prototyp wurde die Anzahl der Kanten, ihr Startknoten und Zielknoten zufällig bestimmt. Dieses Beispiel sollte die wenigen Schritte demonstrieren, die zur Erstellung der Elemente des Graphen notwendig sind. Prefuse ermöglicht einen einfachen Einstieg, die Webseite stellt ein Handbuch und eine API zur Verfügung [Ber13].
Auch von der wissenschaftlichen Seite findet Prefuse Anerkennung, beispielsweise wurde das Framework auf der CHI 2005[10] vorgestellt [HCL05]. Bereits die Informationen der Abhandlung [HCL05] reichen für einen Einstieg in das Framework und dessen Verwendung aus. Prefuse verfügt auch Jahre nach der letzten Änderung über eine aktive Community. Bei Stackoverflow[11] existieren beispielsweise mindestens 61 Fragen zu Prefuse und auch die eigenen Diskussionsforen sind gut besucht[12]. Zu Prefuse existieren zahlreiche Dokumente, die Association for Computing Machinery listet beispielsweise 152 Abhandlungen zu Prefuse. Auch in anderen ähnlichen Projekten kam Prefuse bereits zum Einsatz, beispielsweise in SOVA [BJKK10]. Weitere Verwendung fand Prefuse in mindestens hundert weiteren Projekten[13].
Die Möglichkeiten zur Veränderung des Graphen sind vielfältig gegeben. Beispielsweise kann der Graph direkt bei seiner Generierung an die eigenen Wünsche angepasst werden.

[10] http://www.chi2005.org/index.html
[11] http://stackoverflow.com/questions/tagged/prefuse
[12] http://sourceforge.net/p/prefuse/discussion/343012
[13] https://masterbranch.com/prefuse-projects

In Quelltext 5.6 wurden beispielsweise sämtliche Kanten eine Graue und allen Knoten eine grüne Farbe zugewiesen. Dies geschah durch die beiden Anweisungen „ColorAction fill = new ColorAction("graph.nodes", VisualItem.FILLCOLOR, ColorLib.rgb(0, 200, 0));“ und „ColorAction edges = new ColorAction("graph.edges", VisualItem.STROKECOLOR, ColorLib.gray(200));“.
Der Code von Prefuse ist gut strukturiert und kommentiert, sodass man sich als Entwickler schnell zurechtfindet. Änderungen an Prefuse können leicht durchgeführt und durch integrierte JUnit-Tests geprüft werden. Die Performance von Prefuse ist mehr als ausreichend. Als einziges Framework verfügt Prefuse über einen integrierten Benchmark. Das Resultat des Benchmarks kann Quelltext 5.7 entnommen werden. Pefuse generiert 10000 Elemente auf einem alten „Intel Core 2 Duo“ innerhalb von 3 Sekunden. Die genauen Ergebnisse sind jedoch für diese Beurteilung irrelevant, zumal kein exakter Vergleich mit anderen Frameworks möglich ist. Für den geforderten Anwendungszweck kann Prefuse damit als ausreichend performant angesehen werden.
Es ist gut in die eigene Anwendung integrierbar und die Interaktivität mit dem Graphen ist ebenfalls sehr gut. Der Nutzer kann beispielsweise Knoten mittels drag & drop verschieben oder das Mausrad zum Zoomen verwenden.

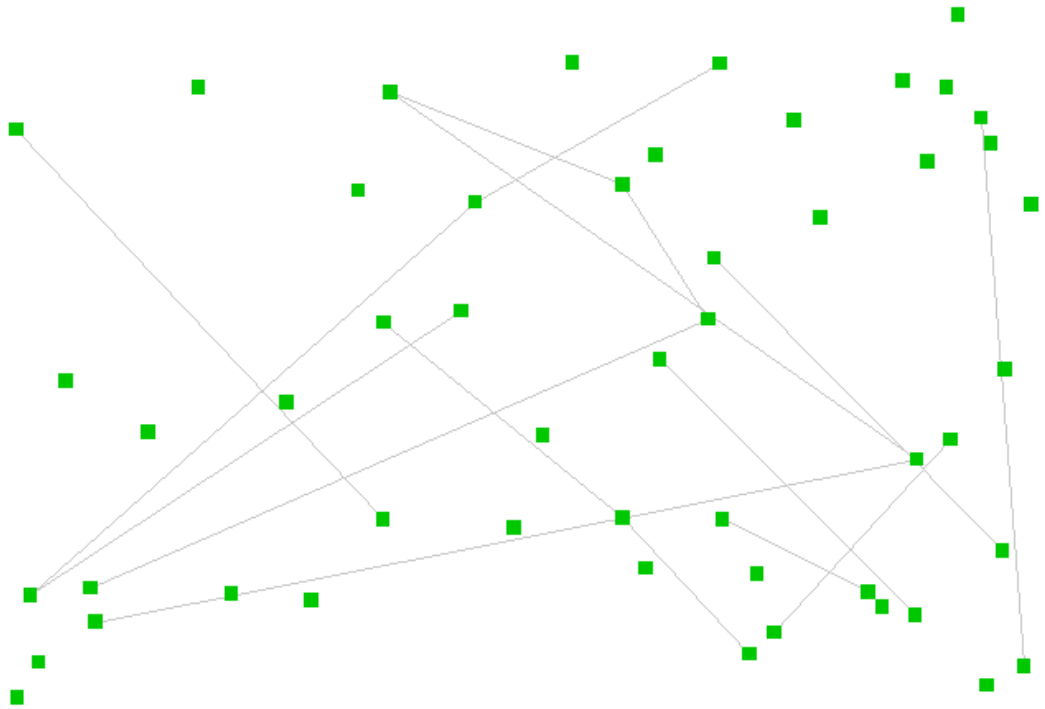


Abbildung 5.13.: Visualisierung des Graphen aus Quelltext 5.6.

Quelltext 5.6 Erstellung eines Demonstrationsgraphen mittels Prefuse.

```
Graph graph = new Graph();
// create Nodes
for (int i=0; i<=50; i++) {
       graph.addNode();
}
// create connections
Random rnd = new Random();
for (int i=0; i<=15; i++) {
       int a = rnd.nextInt(50);
       int b = rnd.nextInt(50);
       graph.addEdge(a, b);
}

ColorAction fill = new ColorAction("graph.nodes", VisualItem.FILLCOLOR,
    ColorLib.rgb(0, 200, 0));
ColorAction edges = new ColorAction("graph.edges", VisualItem.STROKECOLOR,
    ColorLib.gray(200));
ActionList color = new ActionList();
color.add(fill);
color.add(edges);
ActionList layout = new ActionList();
layout.add(new RandomLayout("graph"));
layout.add(new RepaintAction());
Visualization vis = new Visualization();
vis.add("graph", graph);
vis.putAction("color", color);
vis.putAction("layout", layout);
ShapeRenderer r = new ShapeRenderer();
vis.setRendererFactory(new DefaultRendererFactory(r));
Display d = new Display(vis);
d.setSize(720, 500);
d.addControlListener(new DragControl());
d.addControlListener(new PanControl());
d.addControlListener(new ZoomControl());
JFrame frame = new JFrame("prefuse example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(d);
frame.pack();
frame.setVisible(true);
vis.run("color");
vis.run("layout");
```

Quelltext 5.7 Ergebnisse des integrierten Prefuse Benchmarks.

```
Nov 15, 2013 3:32:00 PM test.RenderingBenchmarks paintComponent
INFO: Rendering Benchmarks: NORMAL
PRIMITIVE             COUNT  TIME   PRIMITIVES/SEC PRIMITIVES/FRAME @ 20fps
lines-direct          10000  0.01s  1000000.0 pr/s 50000.0 pr/fr
lines-shape           10000  0.022s 454545.45 pr/s 22727.27 pr/fr
rect-direct-draw      10000  0.023s 434782.6 pr/s  21739.13 pr/fr
rect-shape-draw       10000  0.118s 84745.76 pr/s  4237.28 pr/fr
rect-direct-fill      10000  0.022s 454545.45 pr/s 22727.27 pr/fr
rect-shape-fill       10000  0.042s 238095.23 pr/s 11904.76 pr/fr
rrect-direct-draw     10000  0.12s  83333.33 pr/s  4166.66 pr/fr
rrect-shape-draw      10000  0.086s 116279.06 pr/s 5813.95 pr/fr
rrect-direct-fill     10000  0.255s 39215.68 pr/s  1960.78 pr/fr
rrect-shape-fill      10000  0.251s 39840.63 pr/s  1992.03 pr/fr
text-direct-int       10000  0.039s 256410.25 pr/s 12820.51 pr/fr
text-direct-float     10000  0.035s 285714.28 pr/s 14285.71 pr/fr
text-glyph-vector     10000  0.035s 285714.28 pr/s 14285.71 pr/fr

Nov 15, 2013 3:32:05 PM test.RenderingBenchmarks paintComponent
INFO: Rendering Benchmarks: ANTI-ALIASING
PRIMITIVE             COUNT  TIME   PRIMITIVES/SEC PRIMITIVES/FRAME @ 20fps
lines-direct          10000  0.045s 222222.22 pr/s 11111.11 pr/fr
lines-shape           10000  0.007s 1428571.42 pr/s 71428.57 pr/fr
rect-direct-draw      10000  0.06s  166666.66 pr/s 8333.33 pr/fr
rect-shape-draw       10000  0.007s 1428571.42 pr/s 71428.57 pr/fr
rect-direct-fill      10000  0.029s 344827.58 pr/s 17241.37 pr/fr
rect-shape-fill       10000  0.004s 2500000.0 pr/s 125000.0 pr/fr
rrect-direct-draw     10000  2.36s  4237.28 pr/s   211.86 pr/fr
rrect-shape-draw      10000  1.233s 8110.3 pr/s    405.51 pr/fr
rrect-direct-fill     10000  0.952s 10504.2 pr/s   525.21 pr/fr
rrect-shape-fill      10000  0.774s 12919.89 pr/s  645.99 pr/fr
text-direct-int       10000  0.048s 208333.33 pr/s 10416.66 pr/fr
text-direct-float     10000  0.038s 263157.89 pr/s 13157.89 pr/fr
text-glyph-vector     10000  0.055s 181818.18 pr/s 9090.9 pr/fr
```

5.7. Piccolo2D

Bei Piccolo2D handelt es sich um ein, auf Java basiertes Grafikframework, welches sich selbst als zoombares Userinterface beschreibt. Entwickelt wurde Piccolo2D vom Mensch-Computer Interaktionslabor der Universität von Maryland[14]. Die Visualisierung eines Graphen mittels Piccolo2D kann Abbildung 5.14 entnommen werden. Sie ist als Beispiel direkt im Piccolo2D Quellcode enthalten. Piccolo2D ist kein Framework zur reinen Visualisierung von Graphen, stattdessen lassen sich 2D Strukturen modellieren. Abbildung 5.15 zeigt Piccolo2D beispielsweise bei der Visualisierung von Tabellen, auch dieses Beispiel ist direkt dem Quellcode entnommen. Die Dokumentation zu Piccolo erscheint dürftig, so enthält die Piccolo2D-Webseite

[14] http://www.cs.umd.edu/hcil/jazz/index.shtml

nur eine wenige Seiten umfassende Anleitung zum Einstieg in das Framework. Der Quellcode hingegen demonstriert seine Einsetzbarkeit anhand zahlreicher Beispiele. Aufgrund der Tatsache, dass wichtige Kriterien im Gegensatz zu den anderen vorgestellten Frameworks nicht erfüllt wurden, schied Piccolo2D für weitere Untersuchungen aus.

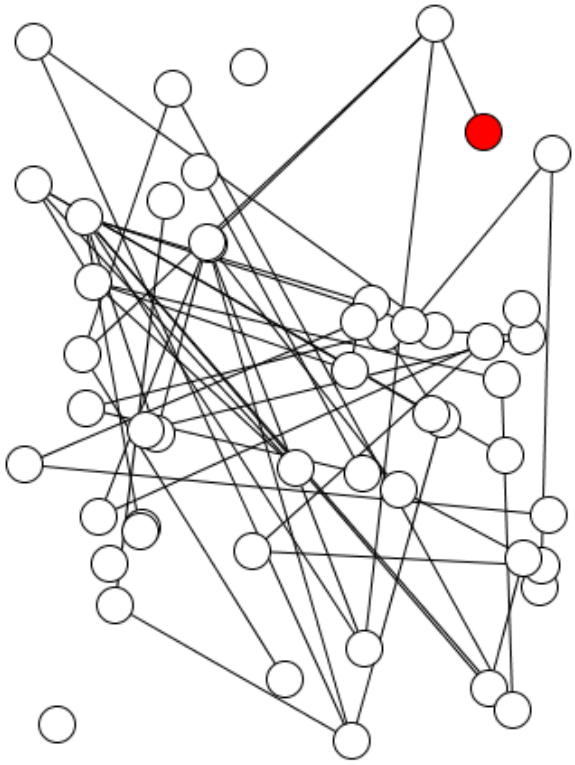


Abbildung 5.14.: Visualisierung eines Graphen mittels Piccolo2D.

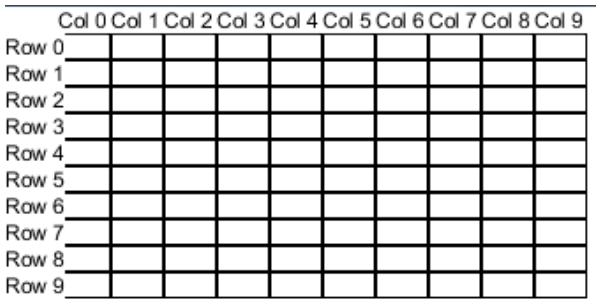


Abbildung 5.15.: Visualisierung eines Graphen mittels Piccolo2D.

5.8. GraphStream

GraphStream ist ebenfalls ein Java basiertes Framework zur Visualisierung von Graphen. Die Webseite von GraphStream[15] enthält eine Demonstration des Frameworks. Abbildung 5.16 zeigt einen mittels GraphStream generierten Graphen, dessen Quellcode Quelltext 5.8 entnommen werden kann. Abbildung 5.17 stellt einen komplexeren Graphen dar, dessen Visualisierung der GraphStream Galerie entnommen wurde [Gra10]. Zwar vermittelt die Webseite von GraphStream einen Eindruck von der Mächtigkeit des Frameworks, scheint sich aber auf die Dokumentation, die auf der Webseite zur Verfügung steht, zu beschränken. Zum Zeitpunkt der Erstellung dieses Dokumentes wurde der Release von GraphStream 1.2 um 11 Monate versäumt[16]. Aufgrund der Tatsache, dass wichtige Kriterien im Gegensatz zu den anderen vorgestellten Frameworks nicht erfüllt wurden, schied GraphStream für weitere Untersuchungen aus.

Quelltext 5.8 Erstellung eines Demonstrationsgraphen mittels GraphStream.

```
Graph graph = new SingleGraph("Test");
graph.addNode("A" );
Node n = graph.getNode("A");
n.setAttribute("weight", 1.5);
n.addAttribute("ui.label", "A");
graph.addNode("B" );
graph.addNode("C" );
graph.addNode("D" );
graph.addEdge("AB", "A", "B");
graph.addEdge("BC", "B", "C");
graph.addEdge("CA", "C", "A");
graph.addEdge("CD", "C", "D");
graph.addEdge("BD", "B", "D");
graph.display();
```

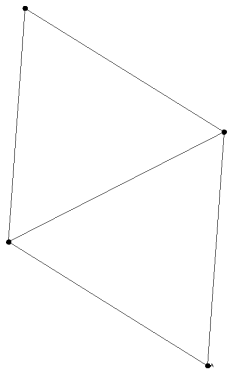


Abbildung 5.16.: Visualisierung des Graphen aus Quelltext 5.8.

[15] http://graphstream-project.org
[16] https://github.com/graphstream/gs-core/issues/milestones

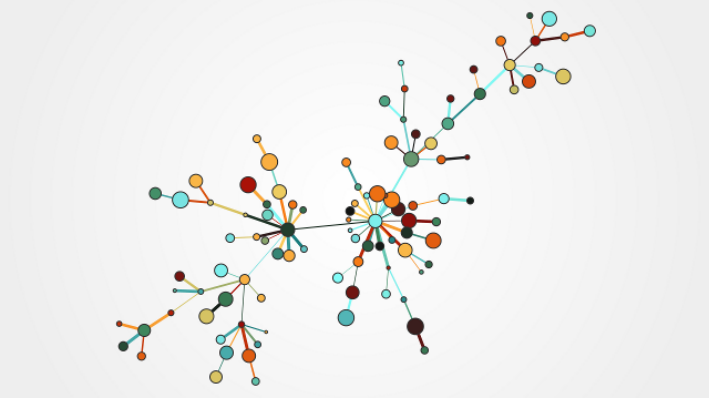


Abbildung 5.17.: Visualisierung eines Graphen, entnommen aus [Gra10].

5.9. Zusammenfassung

Kriterium	GraphViz	Grappa	Zest	JGRaph	JUNG	Prefuse	GraphStream	Piccolo2D
Einarbeitungszeit	+	-	-	+	+	+	+	...
Dokumentation	+	-	-	+	+	+	-	-
Editierbarkeit (Graph)	+	+	-	-	+	+	...	...
Editierbarkeit (Code)	Java: - C: +	-	-	+	+	+	...	...
Performance	+	+	...	+	+	+	...	...
Interaktivität	-	-	+	-	+	+	...	...
Integrierbarkeit	-	-	+	+	+	+	...	...

5.10. Entscheidung

Nach ausführlicher Analyse der vorgestellten Frameworks auf Basis der in diesem Kapitel vorgestellten Kriterien ergeben sich die beiden Favoriten: JUNG und Prefuse. GraphViz verfügt über eine gute Bewertung, die fehlende Interaktivität lässt GraphViz für dieses Projekt aus der näheren Auswahl allerdings ausscheiden.
Prefuse scheint in den Demonstrationsbeispielen gegenüber JUNG optisch ansprechendere Ergebnisse liefern zu können. Prefuse scheint eine aktivere Community zu besitzen. Prefuse wird von den Entwicklern als abgeschlossen betrachtet, die letzte Ergänzung am Code stammt aus dem Jahre 2008. Trotz dieser Zeitspanne scheint die Prefuse-Community weiterhin aktiv zu sein[17]. Des Weiteren scheint die verfügbare Dokumentation zu Prefuse hinsichtlich Umfang minimal besser, als jene für JUNG, zu sein. Aus diesen Gründen wurde im Rahmen dieser Arbeit das Framework Prefuse für die Entwicklung eines geeigneten Visualisierungskonzepts für die kompakte und ganzheitliche Visualisierung von Ontologien gewählt.

[17] http://sourceforge.net/p/prefuse/discussion/343013

6. Implementierung

Nachdem die Grundlagen für die technische Realisierung in den vorherigen Kapiteln erläutert wurden, kann mit der Implementierung begonnen werden.
Realisiert wird die Visualisierung von Ontologien als Plug-in für Protégé. Dadurch entfällt die Notwendigkeit der Implementierung eigener Funktionen zum Laden und Speichern von Ontologien. Des Weiteren kann das Datenmodell der OWL-API verwendet werden, denn Protégé nutzt intern selbst die OWL-API.
Die grafische Darstellung des in Kapitel 4 vorgestellten Konzepts nutzt das in Kapitel 5 ausgewählte Grafikframework Prefuse. Die Verwendung von Prefuse ermöglicht eine Reduzierung des Realisierungsaufwandes, schließlich reduziert sich der Umfang der zu implementierenden Funktionen.
In diesem Kapitel wird zunächst der chronologische Ablauf der Entwicklung beschrieben, um anschließend den Entwurf und einige Designentscheidungen vorzustellen.

6.1. Chronologischer Ablauf

Die Entwicklung des Protégé Plug-ins zur Visualisierung von Ontologien verläuft in drei Etappen. Diese Abschnitte sind in Abbildung 6.1 dargestellt und werden in den folgenden Abschnitten näher erläutert.

Abbildung 6.1.: Visualisierung des chronologischen Ablauf der Entwicklung.

6.1.1. Darstellung der Grundformen

Bereits während der Evaluation des Grafikframeworks Prefuse wurde ein rudimentärer Graph als Prototyp generiert. Dieser Graph aus Abbildung 5.13 wurde im ersten Abschnitt ersetzt. Die im Konzept aus Kapitel 4 erwähnten Abbildungen müssen zerlegt und die Bestandteile des Konzeptes durch Prefuse gerendert werden. Abbildung 4.3 besteht beispielsweise aus mindestens drei verschiedenen, in Abbildung 6.2 dargestellter Kantenformen. Des Weiteren werden Knoten unterschiedlichster Form benötigt. Abbildung 6.3 zeigt das Ergebnis dieser

ersten Realisierungsphase. Dieser Ausschnitt eines Graphen enthält die meisten der benötigten Grundformen. Zur Realisierung dieser Darstellung muss Prefuse erweitert werden. Zwar bietet Prefuse bereits zahlreiche Möglichkeiten zur Modifikation der Graphendarstellung, diese beziehen sich jedoch meist auf den gesamten Graphen. Die beispielsweise in Abbildung 6.4 vorkommende Modifikation vergibt allen Linien und Pfeilspitzen eine schwarze Farbe. Dies ist jedoch nicht gewollt. Für die im Konzept gewünschte Darstellung wird die Möglichkeit benötigt, die Form einzelner Knoten und Kanten gezielt zu verändern. In dieser ersten Etappe geht es vor allem um die Erweiterung von Prefuse, um alle benötigten Grundformen darstellen zu können. Dies geschieht durch Erweiterung der Klassen prefuse.render.EdgeRenderer und prefuse.render.AbstractShapeRenderer. Nach der Erweiterung können einzelne Kanten und Knoten durch zusätzliche Parameter gezielt angepasst werden, um die grafische Darstellung der jeweiligen Parameter kümmern sich die entsprechenden Renderer. Jede Kante und jeder Knoten innerhalb des Prefuse-Graphen enthalten eine eigene Wertetabelle mit Key-Value Paaren. Dies erweist sich als äußerst nützlich, schließlich können die gewünschten Parameter auf diese Weise übergeben werden, Quelltext 6.1 zeigt einen Ausschnitt hiervon. Als Datenquelle für diesen ersten Graphen aus Abbildung 6.3 dienen zufällig generierte Werte. Mit diesen, voneinander unabhängigen, zufälligen Werten sollen alle möglichen Wertekombinationen abgedeckt werden und damit eventuell auftretende Fehler leichter entdeckt werden. Der Ablauf der ersten Etappe entspricht dem Sequenzdiagramm aus Abbildung 6.5.

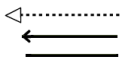


Abbildung 6.2.: In VOWL verwendete Kantenformen.

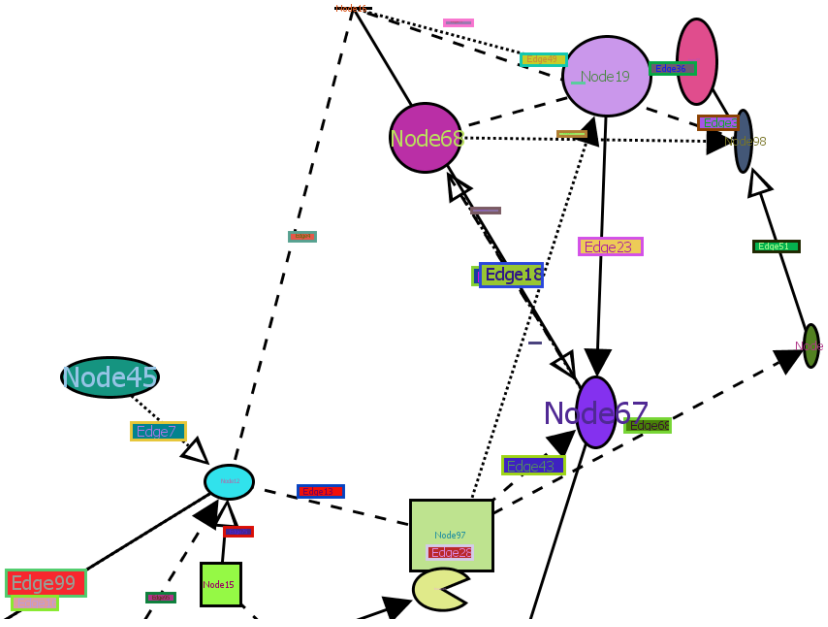


Abbildung 6.3.: Ausschnitt eines Graphen, der alle benötigten Grundformen enthält.

Abbildung 6.4. Änderung der Visualisierung eines Graphen in Prefuse.

```
ColorAction edges = new ColorAction("GraphDataModifier.edges",
    VisualItem.STROKECOLOR, ColorLib.rgb(0, 0, 0));
ColorAction arrow = new ColorAction("GraphDataModifier.edges", VisualItem.FILLCOLOR,
    ColorLib.rgb(0, 0, 0));
```

Quelltext 6.1 Ausschnitt der Datengenerierung eines Knoten.

```
Node n = graph.addNode();
n.set(ColumnNames.ID, rnd.nextInt(25000));
n.set(ColumnNames.NODE_HEIGHT, rnd.nextInt(35));
n.set(ColumnNames.NODE_WIDTH, rnd.nextInt(35));
n.set(ColumnNames.TEXT_SIZE, rnd.nextInt(12));
n.set(ColumnNames.COLOR_RED, rnd.nextInt(250));
n.set(ColumnNames.COLOR_GREEN, rnd.nextInt(250));
n.set(ColumnNames.COLOR_BLUE, rnd.nextInt(250));
n.set(ColumnNames.TEXT_COLOR_RED, rnd.nextInt(250));
n.set(ColumnNames.TEXT_COLOR_GREEN, rnd.nextInt(250));
n.set(ColumnNames.TEXT_COLOR_BLUE, rnd.nextInt(250));
```

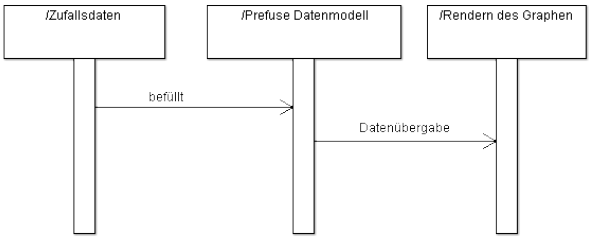


Abbildung 6.5.: Sequenzdiagramm der erste Etappe.

6.1.2. Darstellung der VOWL-Elemente

Nachdem die benötigten Grundformen in 6.1.1 bereits modelliert wurden, konnte mit der Umsetzung des Konzeptes aus Kapitel 4 begonnen werden. Dies geschah durch Ersetzen des in Abschnitt 6.1.1 eingeführten Zufallsdatengenerators. Als Datenquelle des zweiten Abschnitts dient ein zuvor definiertes VOWL-Beispiel. Dieses VOWL-Beispiel wurde durch ein weiteres Modul in das Prefuse-Datenmodell übersetzt und anschließend vom Prefuse Renderer als Graph gezeichnet. Da die hierzu nötigen Modifikationen an Prefuse bereits zuvor erledigt wurden, waren in dieser Phase keine weiteren Erweiterungen an Prefuse notwendig. Das Sequenzdiagramm aus Abbildung 6.6 erläutert den Aufbau in diesem zweiten Abschnitt. Quelltext 6.2 zeigt einen Ausschnitt aus der durch das VOWL-Beispiel definierten Ontologie. Sie wird durch den GraphDataModifier in das Prefuse-Datenmodell übersetzt, um anschließend den Graphen aus Abbildung 6.7 generieren zu können.

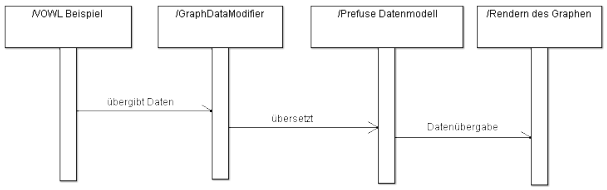


Abbildung 6.6.: Sequenzdiagramm der zweiten Etappe.

Quelltext 6.2 Ausschnitt aus der Generierung des VOWL-Beispiels.

```
GraphStorage.newGraph();
GraphDataModifier mod = new GraphDataModifier();
mod.addClassThing(1);
mod.addClass("Person");
mod.addInstanceToClass("Person", 5);
mod.addClass("Agent");
mod.addInstanceToClass("Agent", 16);
mod.addClass("Document");
mod.addProperty("Agent", Nodetype.vowltype[1], "Agent", Nodetype.vowltype[1]);
mod.addInstanceToClass("Document", 12);
mod.addDeprecatedClass("Spartial Thing");
mod.addInstanceToClass(mod.findClass("Spartial Thing", Nodetype.vowltype[2]), 5);
mod.addProperty("Person", Nodetype.vowltype[1], "Spartial Thing",
    Nodetype.vowltype[2]);
mod.addProperty("Document", Nodetype.vowltype[1], "Agent", Nodetype.vowltype[1]);
```

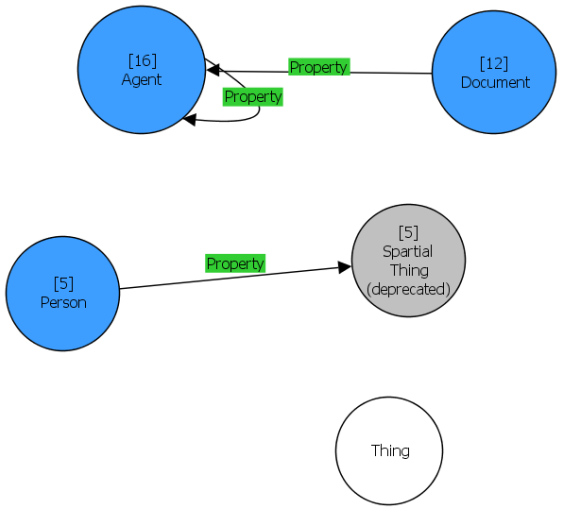


Abbildung 6.7.: Darstellung der VOWL-Elemente innerhalb des Gaphens.

6.1.3. Darstellung der eingelesenen Ontologie

Im dritten und letzten Schritt wird die zuvor fest vorgegebene Ontologie durch eine beliebige, von Protégé eingelesene Ontologie, ersetzt. Das Sequenzdiagramm aus Abbildung 6.8 und die schematische Skizze aus Abbildung 6.9 verdeutlichen den Ablauf in dieser Phase. Auf die eingelesene Ontologie wird mittels der OWL-API zugegriffen. Das Modul TransformOWLtoGraph zerlegt die gespeicherte Ontologie in ihre Klassen und Beziehungen. Diese werden anschließend durch den GraphDataModifier in für Prefuse verständliche Daten übersetzt. Falls die MUTO-Ontologie [LDA11] eingelesen wird, so stellt Abbildung 6.10 das visuelle Resultat dieses Schrittes dar. In diesem letzten Entwicklungsschritt wird ebenfalls die rechts in Abbildung 6.10 sichtbare Informationsleiste implementiert und die innerhalb des Graphen selektierten Elemente optisch sichtbar gekennzeichnet. Die mit der Maus anvisierten Elemente werden ebenfalls farblich markiert.

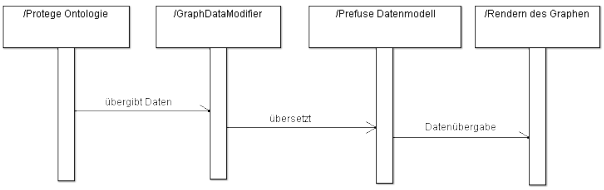


Abbildung 6.8.: Sequenzdiagramm der dritten Etappe.

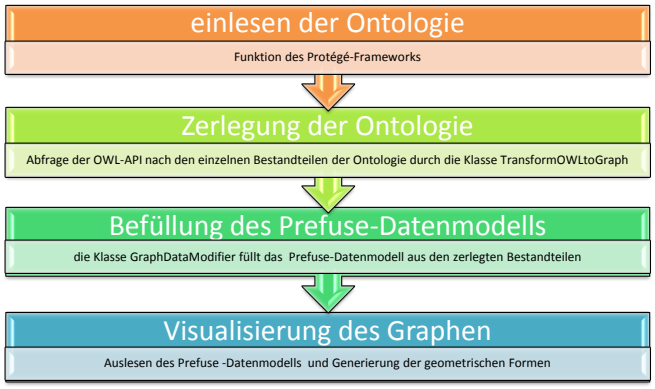


Abbildung 6.9.: Schematische Dastellung der einzelnen Schritte der dritten Etappe.

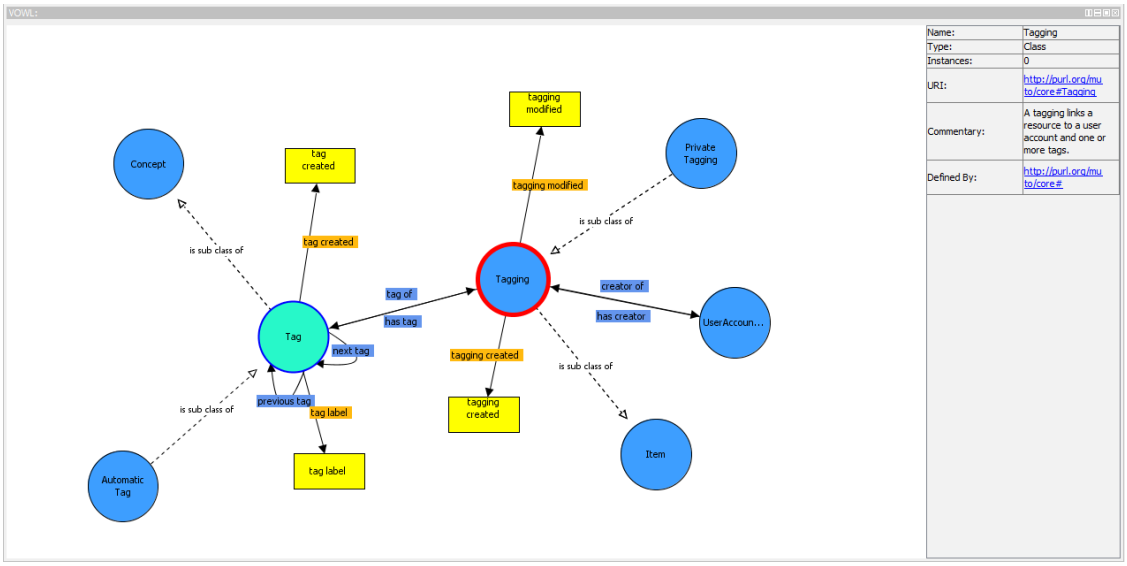


Abbildung 6.10.: Visualisierung der MUTO-Ontologie.

6.2. Architektur

Dieses Kapitel beschreibt die Architektur des Plug-ins. Abbildung 6.11 vermittelt einen groben Überblick über die einzelnen Pakete. Die Funktion und Bedeutung der einzelnen Pakete wird in den folgenden Abschnitten näher erläutert.

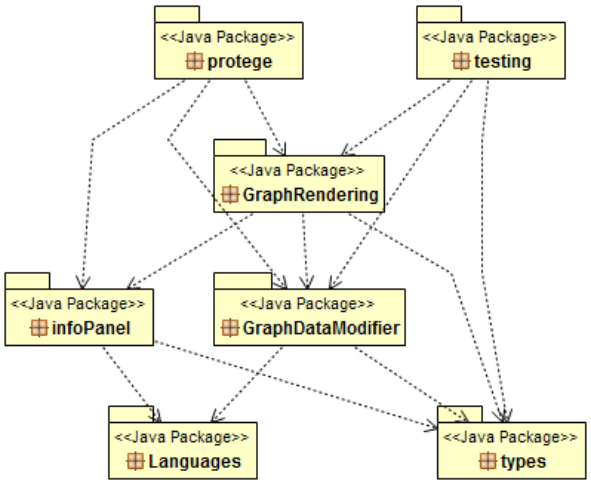


Abbildung 6.11.: Paketansicht des Prototyps.

6.2.1. Das Paket Languages

Abbildung 6.12 enthält eine detailliertere Darstellung des Pakets Languages. Dieses Paket beinhaltet eine Sammlung statischer Strings (static & final), die als Platzhalter der jeweiligen Übersetzung dienen. Auf diese Weise wird eine spätere Übersetzung in andere Sprachen ebenso erleichtert, wie die Unterstützung verschiedener Sprachen. Dieses Paket ist ein Nachbau des Android MultiLanguage Konzeptes[1]. LanguagesGraphEN.java enthält die innerhalb des Grapes verwendeten englischen Begriffe. LanguagesInfoPanelEN.java beinhaltet die in der rechten Informationsleiste verwendeten englischen Begriffe. Quelltext 6.3 enthält einen kleinen Auszug der LanguagesInfoPanelEN.java.

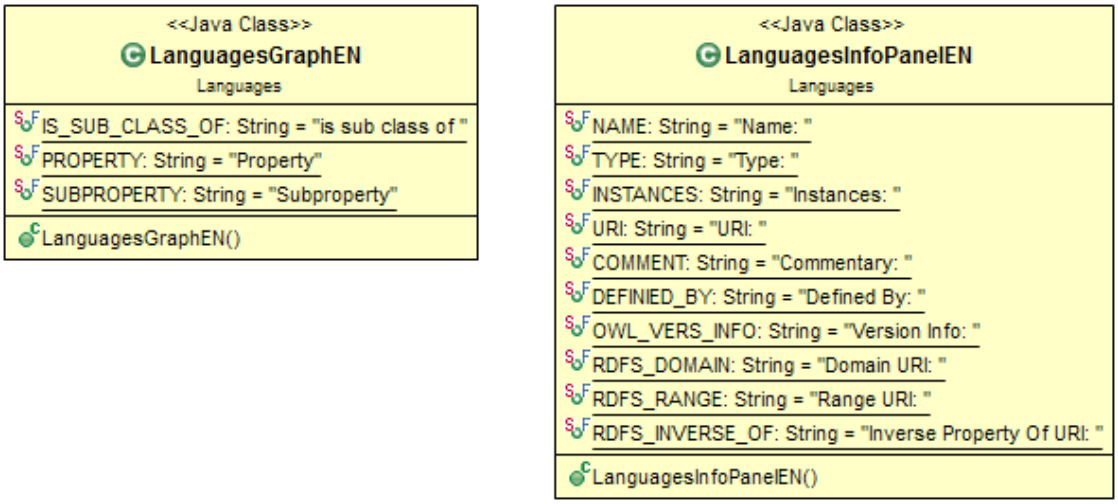


Abbildung 6.12.: UML-Klassendiagramm des Paket Languages.

Quelltext 6.3 Auszug der LanguagesInfoPanelEN.java.

```
public final static String   COMMENT              = "Commentary: ";
public final static String   DEFINIED_BY          = "Defined By: ";
public final static String   OWL_VERS_INFO        = "Version Info: ";
```

6.2.2. Das Paket testing

Abbildung 6.13 zeigt den Aufbau des Pakets testing. Es enthält Module, die hauptsächlich in den Entwicklungsschritten der Abschnitte 6.1.1 und 6.1.2 benötigt werden.

[1] http://developer.android.com/training/basics/supporting-devices/languages.html

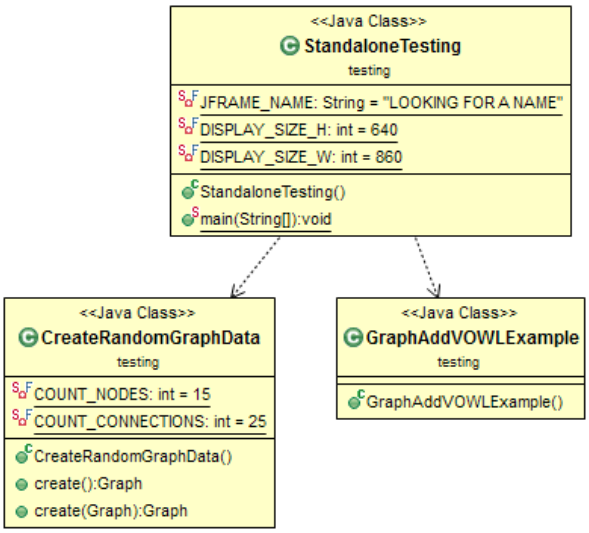


Abbildung 6.13.: UML-Klassendiagramm des Paket testing.

GraphAddVOWLExample

Das Modul GraphAddVOWLExample generiert das zuvor festgelegte VOWL-Beispiel aus dem zweiten Entwicklungsschritt des Abschnittes 6.1.2. Dieses Modell wird anschließend durch den GraphDataModifier des Paketes GraphDataModifier in das Prefuse-Datenmodell übersetzt.

CreateRandomGraphData

Das Modul CreateRandomGraphData wird im ersten Entwicklungsschritt aus Abschnitt 6.1.1 benötigt, um Knoten und Kanten mit Zufallsdaten generieren zu können.

StandaloneTesting

Dieses Modul umgeht Protégé und visualisiert den Prefuse-Graphen direkt. Dies ist vor allem im ersten Entwicklungsschritt aus Abschnitt 6.1.1 hilfreich, denn auf diese Weise kann der Graph direkt aus Eclipse gestartet werden. Dadurch können die üblichen Entwicklungswerkzeuge, wie den Eclipse Debugger, einfacher verwendet werden. Natürlich entfällt dabei die Möglichkeit des Einlesens einer Ontologie, weswegen dieses Modul nur die festgelegte Ontologie aus der Klasse GraphAddVOWLExample bzw. den Graphen aus Zufallsdaten der Klasse CreateRandomGraphData visualisieren kann.

6.2.3. Das Paket protege

Das in Abbildung 6.14 dargestellte Paket protege enthält eine einzige Klasse. VOWLViewComponent.java erweitert die AbstractOWLViewComponent von Protégé und generiert damit die von Protégé angezeigte ViewComponente. Des Weiteren bestimmt diese Klasse das grundlegende Layout des Plug-ins. VOWLViewComponent legt die relative Position und Größe des Prefuse-Graphen und der rechten Informationsleiste fest. Mittels disposeOWLView kümmert sich die VOWLViewComponent ebenfalls um das Schließen des Plug-ins.

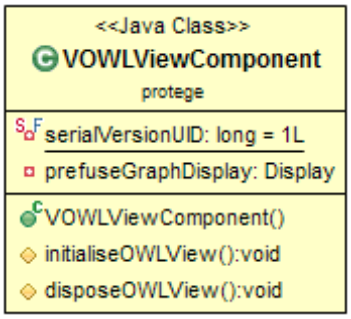


Abbildung 6.14.: UML-Klassendiagramm des Pakets protege.

6.2.4. Das Paket types

Ähnlich zu dem Paket Languages enthält das Paket types statische Strings. Im Gegensatz zu denen des Pakets Languages dienen diese nicht der visuellen Darstellung. Sie werden als Schlüssel in den Key-Value Paaren des Prefuse-Datenmodells verwendet, mit denen die gewünschte Darstellung der entsprechenden Objekte kodiert wird. Die Gruppierung der statischen Strings in dem Paket types soll die Konsistenz der Schreibweise der Key-Value Pair Schlüssel sicherstellen und ein späteres Überarbeiten erleichtern. Nebenbei erhält man auf diese Weise eine Liste aller, im Plug-in verwendeten Schlüssel.Abbildung 6.15 zeigt den Aufbau des Paketes types anhand eines UML-Klassendiagramms.

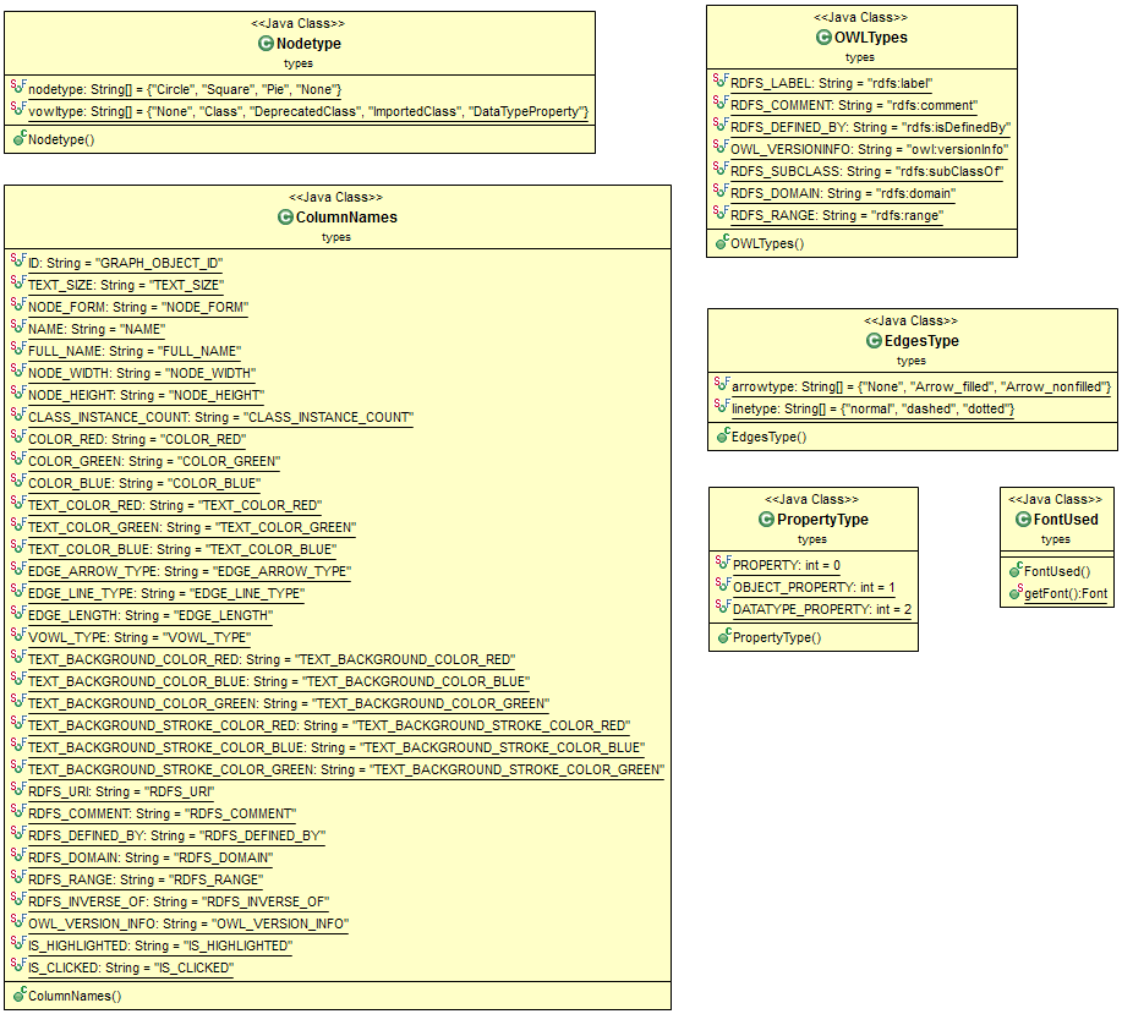


Abbildung 6.15.: UML-Klassendiagramm des Paket types.

PropertyType

Die Klasse PropertyType enthält verschiedene Integer Konstanten zur Unterscheidung der im Konzept (Kapitel 4) definierten Property-Typen: `Object Property`, `Datatype Property` und `Property`. Letztere sind für allgemeine Propertyies aus RDF vorgesehen, die sich weder in `Object Properties` noch in `Datatype Properties` unterscheiden lassen.

OWLTypes

Die Klasse OWLTypes enthält verschiedene statische Strings, die als Bezeichner zum Identifizieren der jeweiligen Attribute innerhalb einer eingelesenen Ontologie dienen. Die OWL-API speichert Bestandteile einer eingelesenen Ontologie als Liste ab. Diese Liste enthält zahlreiche

Klassen des Typs OWLAnnotation, die jeweils durch eine Eigenschaft (getProperty()) und einen Wert (getValue()) repräsentiert werden. Aus diesem Grund erfolgt beim Umwandeln einer eingelesenen Ontologie in einem Graphen ein Vergleich mit den in dieser Klasse gespeicherten Strings. Informationen über das Auslesen der Ontologie aus der OWL-API sind zum Teil in der OWL-API Dokumentation[2] gegeben. Die Zusammenfassung der Strings für einen späteren Vergleich garantiert die Konsistenz der Schreibweise und vermeidet schwer zu überprüfende Fehler, wie beispielsweise Rechtschreibfehler.

Nodetype

Die Klasse Nodetype beinhaltet zwei String-Arrays, die ebenfalls ausschließlich der Beschreibung der Knotenform im Prefuse-Datenmodell dienen. Quelltext 6.4 zeigt verschiedene verwendete Formen. In den Key-Value Paaren des Prefuse-Datenmodells definiert die Klasse Nodetype die möglichen Werte, während die erlaubten Schlüssel durch die Klasse ColumnNames definiert werden. Falls ein Knoten über das Key-Value Pair „NODE_FORM“ und „Circle“ verfügt, so zeichnet der Noderenderer den Knoten als Kreis. Die Größe des Kreises wird beispielsweise durch ein anderes Key-Value Paar definiert.

Quelltext 6.4 Auszug aus der Nodetype.java.

```
{"Circle", "Square", "Pie", "None"}
```

FontUsed

Die Klasse FontUsed dient dem Auslesen der derzeit verwendeten Schrift. Diese Information wird unter anderem von der rechten Informationsleiste benötigt, um den benötigten Platz zum Anzeigen einer Information zu berechnen.

EdgesType

Die Klasse EdgesType definiert die verschiedenen möglichen Kantenformen. Sie enthält zwei String-Arrays, die jeweils die Linienart und die Pfeilspitze definieren. In den Key-Value Paaren des Prefuse-Datenmodells definiert die Klasse EdgesType die möglichen Werte, während die erlaubten Schlüssel durch die Klasse ColumnNames definiert werden. Diese Key-Value Paare werden anschließend vom Edgerenderer ausgelesen.

[2] https://github.com/owlcs/owlapi/wiki/Documentation

ColumnNames

Die Klasse ColumnNames enthält sämtliche möglichen Schlüssel des Key-Value Paares des Prefuse-Datenmodells, die von diesem Plug-in verwendet werden.

6.2.5. Das Paket infoPanel

Abbildung 6.16 zeigt ein UML-Klassendiagramm des Pakets infoPanel. Es stellt die Implementierung der rechten Informationsleiste dar. Die Bestandteile dieses Paketes werden im Folgenden näher erläutert.

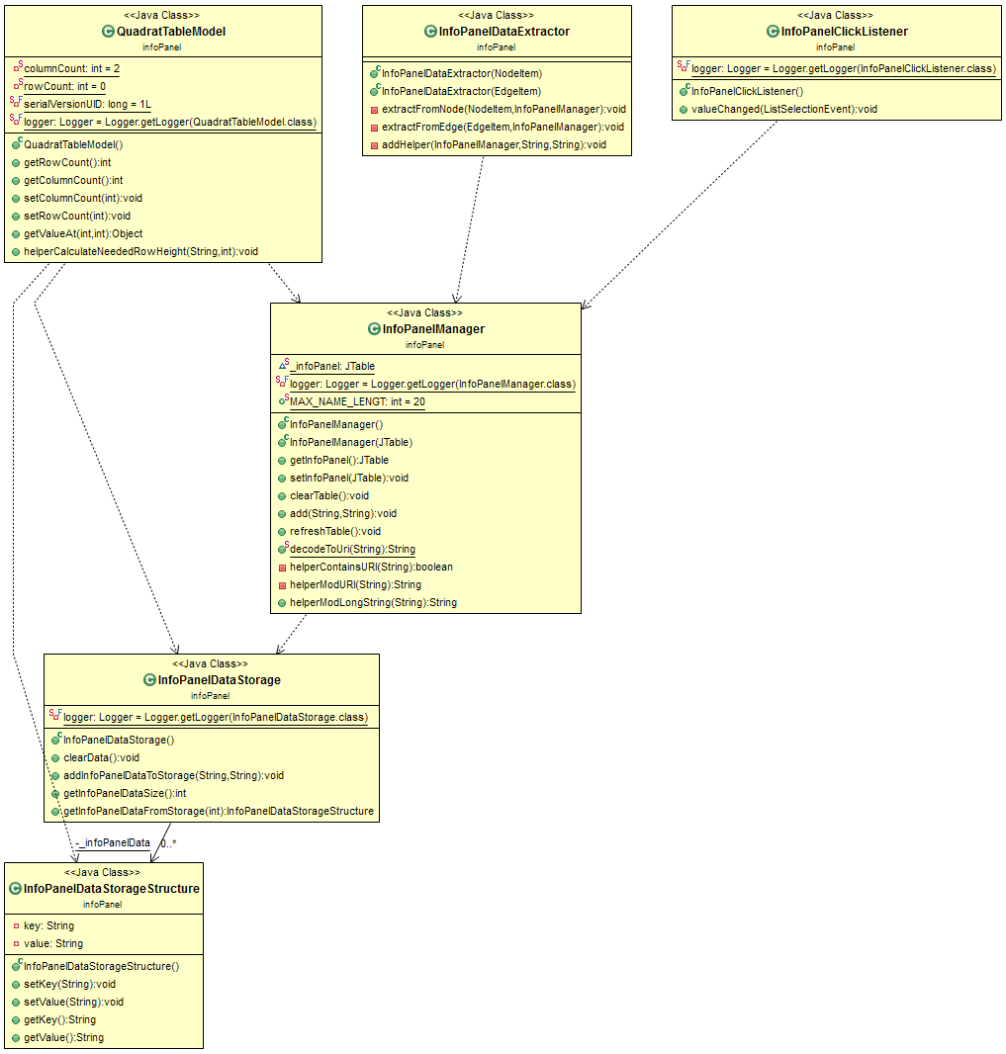


Abbildung 6.16.: UML-Klassendiagramm des Paket infoPanel.

InfoPanelClickListener

Die einzige Funktion des InfoPanelClickListener besteht darin, auf Mauseingaben des Benutzers innerhalb der rechten Informationsleiste zu reagieren. Klickt der Benutzer einen Link innerhalb der Informationsleiste an, so wird dieser mit dem Standardbrowser des Betriebssystems geöffnet. Falls der Nutzer innerhalb der Informationsleiste die Maus verwendet, um mehrere Links gleichzeitig zu selektieren, so werden sämtliche selektierte Links durch den InfoPanelClickListener an den Browser des Betriebssystems zum Öffnen übergeben.

InfoPanelDataExtractor

Falls der Benutzer ein Element des Graphen mit der Maus anklickt, so werden dessen Informationen in der Informationsleiste angezeigt. Der ControlListener aus dem Paket GraphRendering reagiert auf das Klick-Event und dieser übergibt dem InfoPanelDataExtractor das selektierte Element. Die Aufgabe des InfoPanelDataExtractor besteht darin, die notwendigen Informationen über den Zugriff auf das Prefuse-Datenmodell zu extrahieren. Die zusammengestellten Informationen werden in der InfoPanelDataStorage abgespeichert. Der InfoPanelDataExtractor bestimmt, welche Daten extrahiert und im Datenmodell abgelegt werden. Auf diese Weise regelt der InfoPanelDataExtractor welche Daten innerhalb der Informationsleiste angezeigt werden sollen und welche nicht. Sollen weitere Informationen in der Informationsleiste angezeigt werden, so muss der InfoPanelDataExtractor erweitert werden, damit die benötigten Informationen extrahiert und anschließend im InfoPanelDataStorage abgelegt werden.

InfoPanelDataStorage

Das InfoPanelDataStorage stellt das Datenmodell der rechten Informationsleiste dar. Sobald der Nutzer ein Element des Graphen mit der Maus auswählt, wird das InfoPanelDataExtractor mit Daten befüllt. Das InfoPanelDataStorage ist ebenso wie das Prefuse-Datenmodell als Liste von Key-Value Paaren aufgebaut. Diese Key-Value Paare sind Objekte der Klasse InfoPanelDataStorageStructure. Abbildung 6.17 stellt diesen Sachverhalt grafisch dar.

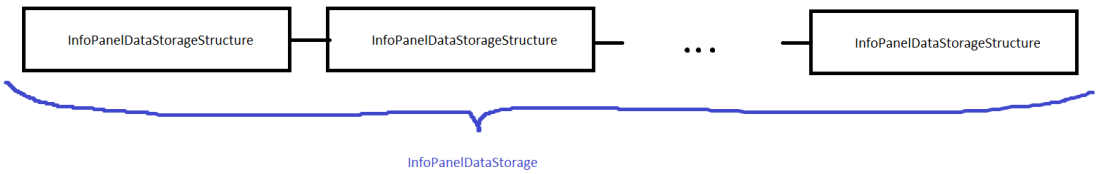


Abbildung 6.17.: Skizze der Datenstruktur.

InfoPanelDataStorageStructure

Das InfoPanelDataStorageStructure regelt die Struktur der Daten des InfoPanels. Die Datenstruktur des InfoPanelDataStorage besteht aus einer Liste von InfoPanelDataStorageStructure Elementen.

QuadratTableModel

Das QuadratTableModel erweitert das AbstractTableModel und stellt Methoden zur Verfügung mit denen die JTable auf das Datenmodell zugreifen kann. Des Weiteren formatiert das QuadratTableModel Links und berechnet die benötigte Höhe einer Zeile.

InfoPanelManager

Der InfoPanelManager enthält Methoden, mit denen Informationen im Datenmodell der Informationsleiste abgelegt werden können. Der InfoPanelDataExtractor verwendet den InfoPanelManager zum Ablegen der erforderlichen Informationen. Des Weiteren dekodiert er Links für den InfoPanelClickListener. Der InfoPanelManager dient ebenfalls als Zugriffspunkt auf die JTable der rechten Informationsleiste. Diese wird beim Generieren des Layouts durch die Klasse VOWLViewComponent benötigt.

6.2.6. Das Paket GraphDataModifier

Das Paket GraphDataModifier enthält sämtliche Module, die zum Ändern der Daten des Prefuse-Datenmodells notwendig sind. Ein UML-Klassendiagramm des Pakets GraphDataModifier ist in Abbildung 6.18 abgebildet. Die einzelnen Bestandteile dieses Paketes werden in den folgenden Abschnitten näher erläutert.

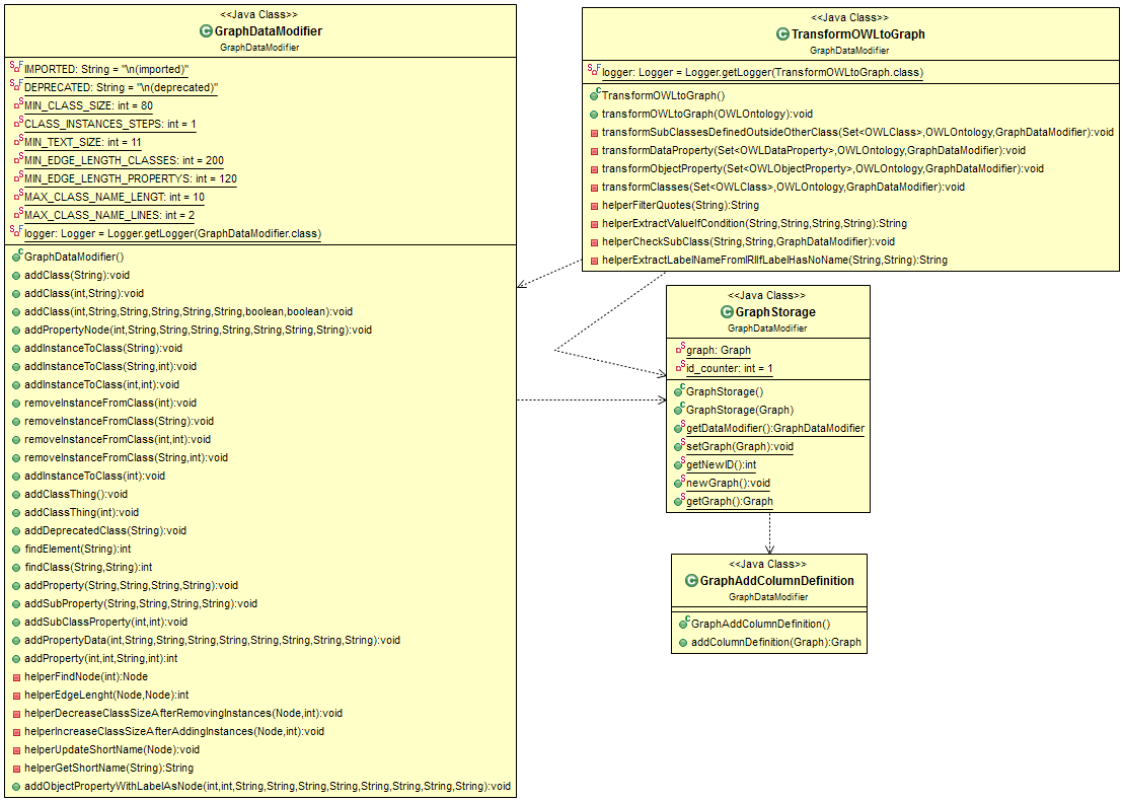


Abbildung 6.18.: UML-Klassendiagramm des Pakets GraphDataModifier.

GraphAddColumnDefinition

Beim Erstellen eines neuen Prefuse-Graphen legt die Klasse GraphAddColumnDefinition alle Schlüssel der Prefuse Datentabelle und damit die Schlüssel der Key-Value Paare fest. Als Schlüssel dienen die statischen Strings der Klasse ColumnNames aus dem Paket types. Die Klasse GraphAddColumnDefinition kommt daher nur bei der Generierung eines Graphen zur Anwendung. Dies ist beispielsweise beim Laden einer Ontologie in Protégé der Fall.

TransformOWLtoGraph

Nachdem eine Ontologie in Protégé eingelesen wurde, extrahiert die Klasse TransformOWLtoGraph sämtliche notwendigen Informationen aus der Ontologie und legt diese Daten im Prefuse-Datenmodell ab. TransformOWLtoGraph extrahiert die notwendigen Informationen, die anschließend von der Klasse GraphDataModifier in das Prefuse-Datenmodell umgewandelt werden. Das Extrahieren der notwendigen Informationen erfolgt durch Zugriff auf die OWL-API. Hierbei wird auch auf die statischen Strings der Klasse OWLTypes des Pakets types zugegriffen.

GraphDataModifier

Die Klasse GraphDataModifier enthält Methoden, mit deren Hilfe OWL-Elemente in das Prefuse-Datenmodell umgewandelt werden. Beim Laden einer Ontologie werden die hierfür benötigen Informationen von der Klasse TransformOWLtoGraph aus der Ontologie der OWL-API extrahiert.

GraphStorage

Die Klasse GraphStorage hält den Prefuse-Graphen und bietet Methoden, um auf diesen zugreifen zu können. Der Prefuse-Graph innerhalb dieser Klasse ist statische, schließlich kann von diesem Plug-in lediglich ein einzelner Graph gleichzeitig visualisiert werden. Dadurch, dass der Graph statisch ist, kann jede andere Klasse des Plug-ins auf den Graphen zugreifen. Dies erleichtert jeder Klasse das Extrahieren benötigter Informationen. Des Weiteren erhält der GraphStorage einen ID-Generator, durch den eindeutige IDs für die weitere Verwendung im Rahmen des Plug-ins generiert werden können.

6.2.7. Das Paket GraphRendering

Das Paket GraphRendering enthält Klassen, die zur Visualisierung des Graphen notwendig sind. Abbildung 6.19 stellt ein UML-Klassendiagramm des Pakets dar. Die einzelnen Bestandteile des Paketes werden in den folgenden Abschnitten näher erläutert.

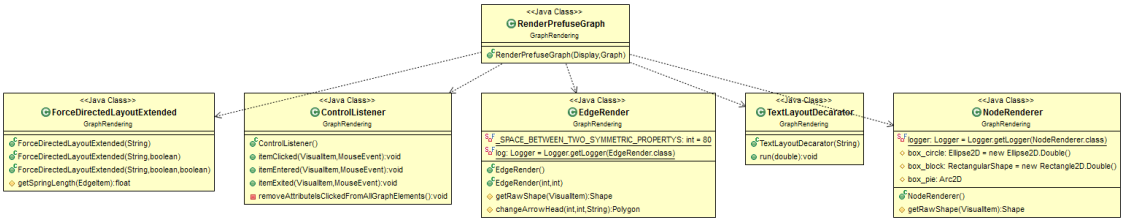


Abbildung 6.19.: UML-Klassendiagramm des Paket GraphRendering

ControlListener

Die Klasse ControlListener erweitert den ControlAdapter und implementiert die abstrakte Klasse Control. Sie reagiert auf Klickeingaben des Benutzers und übergibt diese dem InfoPanelClickListener des infoPanel Pakets. Dabei setzt der ControlListener ein IS_CLICKED Boolean Flag bei dem entsprechenden Element des Graphen auf den Wert „true“. Des Weiteren reagiert der ControlListener auf Mouseover-Ereignisse und setzt dabei das IS_HIGHLIGHTED Flag auf den Wert „true“. Wird ein anderes Element angeklickt bzw. verlässt die Maus

den Bereich des Elementes, so werden die jeweiligen Flags entfernt. Die Visualisierung beider Ereignisse wird, abhängig des selektierten Elements, von dem EdgeRender bzw. dem NodeRenderer übernommen.

TextLayoutDecarator

Die Klasse TextLayoutDecarator regelt das Layout des Textes einer Kante. Hierzu erweitert TextLayoutDecarator das Prefuse Layout. TextLayoutDecarator bestimmt die Position des Textes einer Kante und regelt das Aussehen des Textes. Beispielsweise stellt der TextLayoutDecarator den Text in einer anderen Farbe dar, falls die zugehörige Kante durch den Benutzer markiert wurde. Neben der Position eines Textes regelt der TextLayoutDecarator auch die Textfarbe und die Hintergrundfarbe eines Textes.

ForceDirectedLayoutExtended

Die Klasse ForceDirectedLayoutExtended erweitert das ForceDirectedLayout von Prefuse und regelt die Länge einer Kante des Graphen. Auf diese Weise könnten unterschiedliche Kanten eine verschiedene Länge aufweisen.

EdgeRender

Die Klasse EdgeRender erweitert den Prefuse EdgeRenderer. Sie regelt das Aussehen einer Kante. Hierzu werden die Informationen des Prefuse-Datenmodells ausgelesen und die gewünschte Kantenform generiert. Auf diese Weise können unterschiedliche Pfeilformen innerhalb eines Graphen realisiert werden. Des Weiteren enthält die Klasse EdgeRender eine Sonderbehandlung für symmetrische Properties. Eine Kante zwischen dem Knoten A und dem Knoten A ist in Prefuse korrekterweise sehr kurz. Dieses Verhalten war jedoch nicht erwünscht, denn das Konzept spezifiziert die Form aus Abbildung 6.20 für derartige Kanten. Falls mehrere symmetrische Properties vorhanden sind, so sollen diese, wie Abbildung 6.21 demonstriert nebeneinander platziert werden. Des Weiteren muss der EdgeRender die entsprechenden Flags, gesetzt durch den ControlListener, beachten und markierte und selektierte Kanten in einer anderen Farbe darstellen.

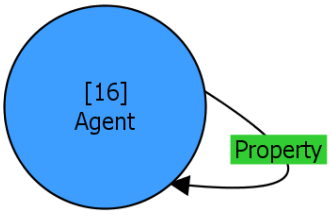


Abbildung 6.20.: Erforderliches Aussehen von symmetrischen Properties.

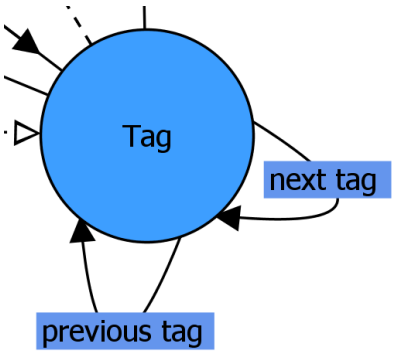


Abbildung 6.21.: Darstellung mehrere symmetrische Properties.

In Version 1.0 von VOWL ist die Existenz mehrere Kanten zwischen Knoten A und B nur indirekt spezifiziert. Dies wird in Abbildung 6.22 verdeutlicht. Der Prefuse Kantenrenderer zeichnet derartige Knoten übereinander. Eine Kante zwischen Knoten A und B und eine Kante zwischen B und A haben dieselben Koordinaten, lediglich ihre Pfeilspitzen unterscheiden sich. Das Beispiel aus Abbildung 6.23 verdeutlicht, dass die derzeit markierte Kante zwischen „Document“ und „Agent“ über der Kante zwischen „Agent“ und „Document“ gezeichnet wird. Beide Kanten können nur über ihre Pfeilspitze unterschieden werden.

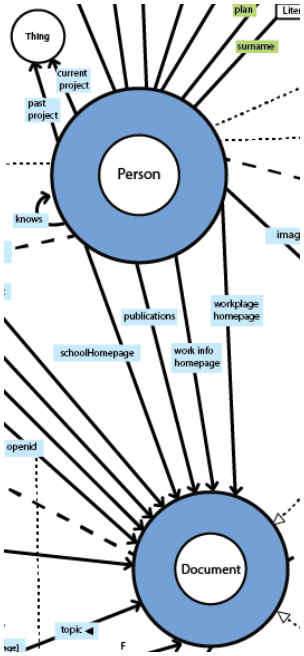


Abbildung 6.22.: Ausschnitt der Konzeptansicht aus VOWL 1.0 [NL13].

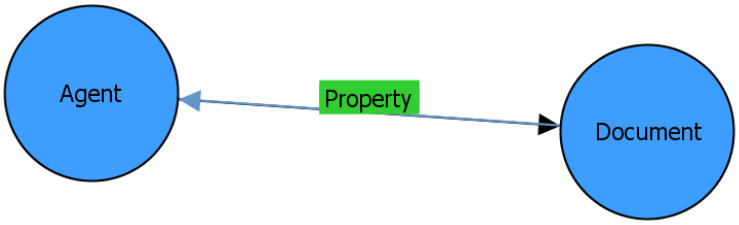


Abbildung 6.23.: Visualisierung der Kante AB und der Kante BA.

Auch dieses Problem wird vom EdgeRender des Plug-ins gelöst. Eine erste Lösung sah ein ausklappbares Dropdown-Menü vor, über das der Benutzer die gewünschte Kante und damit die erwünschte Property auswählen konnte. Diese Idee wurde jedoch verworfen, jeder Bestandteil der Ontologie sollte visuell repräsentiert werden, andernfalls wäre das Erlernen einer unbekannten, großen Ontologie durch mangelnde visuelle Unterstützung unnötig erschwert. Ein zweiter Lösungsansatz sah die Auftrennung einer Kante in zwei Kanten vor. Bei diesem Ansatz sollte das Label der Kante als Knoten zwischen beiden Kanten eingefügt werden. Abbildung 6.24 verdeutlicht diesen zweiten Lösungsansatz.

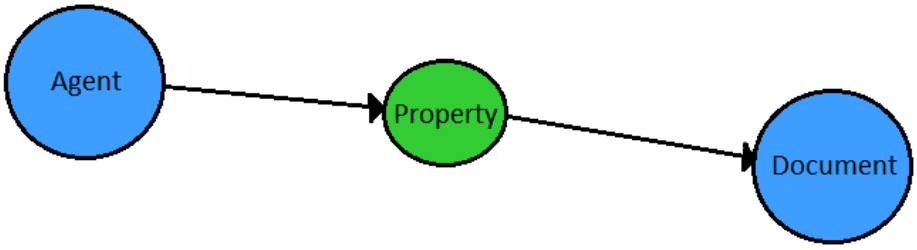


Abbildung 6.24.: Erhoffter Lösungsansatz des Problems aus Abbildung 6.23.

Der zweite Lösungsansatz aus Abbildung 6.24 hatte das visuell nicht ansprechende Ergebnis aus Abbildung 6.25 zur Folge. Die Abstoßungskräfte der nicht verbundenen Zwischenknoten „creator“ und „has_creator“ des kräftebasierten Layouts lassen beide Knoten zu weit voneinander weg driften. Dasselbe Verhalten ist bei „has_tag“ und „tag_of“ zu erkennen.

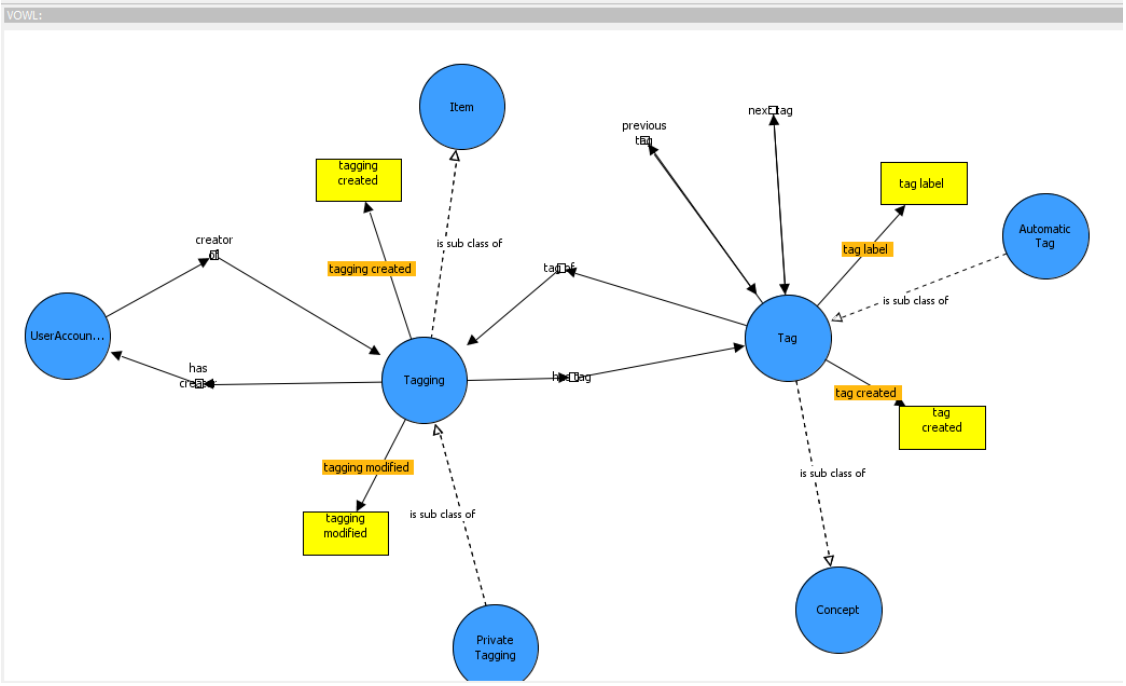


Abbildung 6.25.: Visuell nicht ansprechendes Ergebnis des Lösungsansatzes aus Abbildung 6.23.

Aus diesem Grund wurde ein dritter Ansatz entwickelt, bei dem der EdgeRender des Plug-ins jede weitere Kanten mit einem versetzen Mittelpunkt visualisiert. Dies wird in Abbildung 6.26 visuell dargestellt.

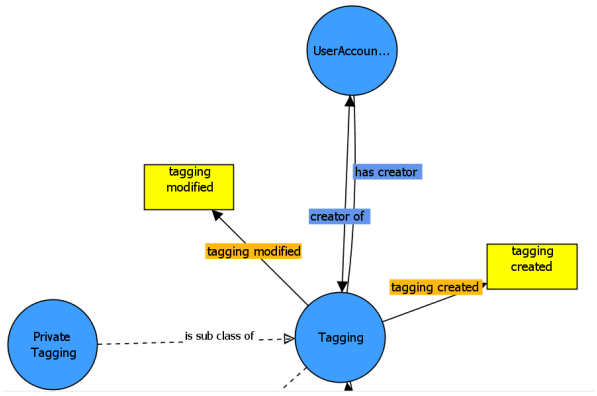


Abbildung 6.26.: Dritter Lösungsansatz.

NodeRenderer

Die Klasse NodeRenderer erweitert den Prefuse AbstractShapeRenderer und bietet Unterstützung für unterschiedliche Knotenformen. Durch die Realisierung der unterschiedlichen Knotenformen können Knoten ein und desselben Graphen eine unterschiedliche Form annehmen. Hierzu werden die Daten des Prefuse-Datenmodells ausgelesen und die entsprechenden Attribute ausgewertet. Der NodeRenderer setzt beispielsweise die Farbe eines Knoten und verändert diese, falls die entsprechenden Flags des ControlListener gesetzt wurden.

RenderPrefuseGraph

Die Klasse RenderPrefuseGraph rendert den Prefuse-Graphen im, durch die VOWLViewComponent des Paketes protege, vorgegebenen Display.

7. Evaluation

Ihm Rahmen dieser Arbeit wurde das in Kapitel 4 vorgestellte und in Kapitel 6 realisierte Visualisierungskonzept durch eine Nutzerstudie evaluiert. Dabei wurde sowohl die Verständlichkeit der Visualisierung als auch der Umgang mit dem Visualisierungswerkzeug untersucht.
Innerhalb der Nutzerstudie wurde die Referenzimplementierung, die das Konzept aus Kapitel 4 implementiert, mit einem weiteren bereits existierenden Visualisierungswerkzeug verglichen. Hierbei handelte es sich um das schon in Abschnitt 3.2 beschriebene SOVA. Als Blindstudie wurde die Nutzerstudie nicht durchgeführt, schließlich konnten die Studienteilnehmer den Namen der jeweiligen Visualisierung ablesen. Die meisten der Teilnehmer der Studie kannten weder die Referenzvisualisierung SOVA noch das Visualisierungskonzept VOWL oder dessen Prototyp. Weiter wussten sie nicht, welche der beiden Visualisierungen zu untersuchenden VOWL-Prototyp entsprach. Damit kam die Studie für die meisten der Teilnehmer den Kriterien einer „einfachblind Studie“ hinsichtlich dieses Kriteriums sehr nahe.
Zum Vergleich des Visualisierungskonzeptes VOWL und dessen prototypischer Umsetzung wurde das Visualisierungswerkzeug SOVA aus der Liste aller themenverwandten Arbeiten (Abschnitt 3) ausgewählt. Die Vergleichsvisualisierung sollte auf demselben Framework basieren, damit dieselbe Ontologie visualisiert werden konnte. Protégé 3 kann Protégé 4 Ontologien nicht fehlerfrei einlesen. Des Weiteren sollte ausgeschlossen werden, dass die Nutzer VOWL aufgrund des moderneren Designs von Protégé bevorzugten. Da OntoGraf im Vergleich zu SOVA zu viele Nachteile aufwies, wurde SOVA als Vergleichsvisualisierung bestimmt.
Durchführung und Resultate der Nutzerstudien werden in diesem Kapitel beschrieben.

7.1. Durchführung

In diesem Abschnitt wird die Durchführung der Benutzerstudie beschrieben. Die Verifikation der prototypischen Umsetzung des Visualisierungskonzeptes aus Kapitel 4 erfolgte als Expertenstudie. Der Prototype wurde von jeweils einer Person unter Anleitung des Studienleiters in einem neutralen Raum validiert.
Die Studie besteht aus einer Nutzerbefragung und einem Laborexperiment. Zu Beginn der Studie erhielten die Teilnehmer einen Fragebogen zur Erfassung ihres Vorwissens. Die enthaltenen Fragen beschränkten sich auf die Erhebung der Erfahrung hinsichtlich der Visualisierung von Ontologien im allgemeinen und den untersuchten Visualisierungen und Ontologien im Speziellen.
Im Anschluss sollten die Studienteilnehmer in einem Laborexperiment verschiedene Aufgaben mit dem Prototyp erledigen. Zur Kontrolle wurden weitere Fragen gestellt, um die

Ergebnisse mit SOVA aus Abschnitt 3.2 als Referenzwerkzeug vergleichen zu können. Die Reihenfolge der Visualisierungsplugins war alternierend, dem ersten Teilnehmer wurde zuerst VOWL anschließend SOVA gezeigt, während die Reihenfolge im Anschluss an den vorherigen Teilnehmer getauscht wurde. Der Studienleiter notierte die Bemerkungen der Probanden und deren Vorgehen.
Abschließend wurde abermals eine Nutzerbefragung durchgeführt, in der die Probanden um ihre abschließende Meinung hinsichtlich der beider Visualisierungen gebeten wurden. Auch wurden den Probanden zwei Abbildungen gezeigt, die sie beschreiben und bewerten sollten.

7.2. Aufgaben

Während des Laborexperiments sollten die Studienteilnehmer 16 Aufgaben lösen. Die ersten acht Aufgaben bezogen sich auf die modular-unified-tagging-ontology (MUTO) [LDA11], die restlichen auf die friend-of-a-friend (FOAF) Ontologie [DB10]. Die MUTO-Ontologie wurde als Referenz für eine kleine Ontologie ausgewählt, während die FOAF-Ontologie als Vertreter einer größeren Ontologie diente.
Jeweils vier der acht Fragen zur MUTO-Ontologie mussten mithilfe des Visualisierungskonzeptes SOVA beantwortet werden, die restlichen vier Fragen unter Zuhilfenahme des Visualisierungskonzeptes VOWL. Mit den acht Aufgaben zur FOAF-Ontologie wurde äquivalent verfahren. Die Reihenfolge der verwendeten Visualisierungskonzepte war alternierend.
Abbildung 3.7 visualisiert die MUTO-Ontologie mithilfe des in Abschnitt 3.2 vorgestellten Visualisierungskonzepts SOVA. Abbildung 6.10 zeigt eine Visualisierung der MUTO-Ontologie mithilfe des in Abschnitt 4 vorgestellten und verbesserten Visualisierungskonzepts. Eine Visualisierung der FOAF-Ontologie durch beide Visualisierungskonzepte befindet sich in Abschnitt B des Anhangs unter Abbildung B.1 und Abbildung B.2. Vergleichbare Darstellungen erhielten die Probanden während der durchgeführten Evaluation.
In diesem Abschnitt werden die einzelnen Fragen und Aufgaben sowie deren Motivation erläutert. Der Fragebogen ist in Abschnitt A des Anhangs beigefügt.
Die einzelnen Fragen sind mit einer dreistelligen Nummer gekennzeichnet. Die erste Ziffer gibt die zu bearbeitende Ontologie an, dabei steht eins für die MUTO-Ontologie und zwei für die FOAF-Ontologie. Die zweite Ziffer dient der Unterscheidung des zu verwendeten Visualisierungswerkzeuges, während die dritte Ziffer die eigentlichen Fragen kennzeichnet. 111 steht beispielsweise für die erste Ontologie (MUTO-Ontologie), das erste Visualisierungswerkzeug und die erste Frage.

7.2.1. Fragen zur MUTO-Ontologie

In Aufgabe 111 wurden die Probanden gebeten, den Namensraum eines Objekts mit dem Namen „Item“ zu bestimmen.
Aufgabe 112 war ähnlich zu 111, nur dass diesmal nach dem Namensraum des Objekts „tag of“ gefragt wurde. In beiden Fällen wurde den Probanden nicht mitgeteilt, ob eine Klasse

oder Property gesucht wurde. Beide Fragen dienten dem Einstieg und sollten die Interaktion des Nutzers mit dem Werkzeug überprüfen.
In **Aufgabe 113** sollte der Nutzer die Anzahl der Klassen bestimmen. Durch diese Frage sollte bestimmt werden, ob die Nutzer Klassen intuitiv als solche bestimmen können.
Durch **Aufgabe 114** wurden die Nutzer schließlich gebeten, das Objekt „Tagging“ zu beschrieben. Durch diese Frage sollte die Visualisierung als Ganzes überprüft werden. Waren die Nutzer in der Lage die Ontologie richtig zu deuten, konnten sie erkennen, welche Richtung durch die Pfeile propagiert werden sollte? Waren sie in der Lage die verschiedenen Properties richtig zu erkennen?
Anschließend wurde das Visualisierungswerkzeug gewechselt und der Nutzer musste in **Aufgabe 121** den Namensraum des Objekts „Concept“ bestimmen.
In **Aufgabe 122** sollten sie den Namensraum des Objekts „nextTag“ herausfinden. Beide Aufgaben haben eine ähnliche Fragestellung wie Aufgabe 111 bzw. 112 und dienen dem Vergleich beider Visualisierungskonzepte.
In **Aufgabe 123** sollte der Nutzer die Anzahl der Properties bestimmen. Diese Frage diente analog zu Frage 123 der Feststellung, ob Properties durch den Nutzer intuitiv als solche zu erkennen seien.
Durch **Aufgabe 124** wurden die Nutzer schließlich aufgefordert, das Objekt „Tag“ zu beschreiben. Analog zu Frage 114 sollte überprüft werden, ob die Nutzer die Visualisierung als Ganzes verstehen konnten. Waren die Nutzer in der Lage die Ontologie richtig zu deuten, konnten sie erkennen, welche Richtung durch die Pfeile propagiert werden sollte? Waren sie in der Lage die verschiedenen Properties richtig zu deuten?

7.2.2. Fragen zur FOAF-Ontologie

Die Fragen zur FOAF-Ontologie waren analog zu den Fragen der MUTO-Ontologie und dienten dem weiteren Vergleich beider Visualisierungskonzepte für größere Graphen. In **Aufgabe 211** und **Aufgabe 221** sollte der Namensraum einer Klasse, in **Aufgabe 212** und **Aufgabe 222** der einer Property bestimmt werden. **Aufgabe 221** stellte eine Besonderheit dar, es existieren zwei Klassen mit demselben Namen „Person“. Durch **Aufgabe 213** und **Aufgabe 223** sollte bestimmt werden, ob der Nutzer innerhalb des Visualisierungskonzeptes Klassen und Properties intuitiv unterscheiden kann und mittels **Aufgabe 214** und **Aufgabe 224** sollte die Visualisierung als Ganzes verifiziert werden.

7.2.3. Abschlussfragen

Im Anschluss an die 16 Aufgaben wurden die Probanden um ihre abschließende Meinung zu VOWL und SOVA gebeten. Sie sollten sowohl die Vorteile als auch die Nachteile nennen, die Visualisierung mittels Schulnoten bewerten und die Schwierigkeit des Auslesens von Klassen, Properties und von Details bewerten. Anschließend wurden den Studienteilnehmern zwei verschiedene Darstellungen gezeigt und sie gebeten, die gezeigte Darstellung zu beschreiben und ihr favorisiertes Konzept zu nennen.

7.3. Studienteilnehmer

An der Nutzerstudie haben sechs Probanden teilgenommen. Alle Teilnehmer kamen aus dem Bereich der Fakultät 5 der Universität Stuttgart und waren männlich.
Das Vorwissen der Teilnehmer war gering, allen Teilnehmern war die FOAF-Ontologie unbekannt, einer von sechs Teilnehmern kannte die MUTO-Ontolgie. Vier von sechs Teilnehmern gaben einen mittleren Kenntnisstand hinsichtlich Ontologien im Allgemeinen an. Vier von sechs Teilnehmern kannten das Visualisierungskonzept VOWL nicht, keiner der sechs Teilnehmer kannte das Visualisierungskonzept SOVA. Drei der Sechs Teilnehmer gaben an, bereits zuvor Werkzeuge zur Visualisierung von Ontologien verwendet zu haben. Abbildung 7.1, Abbildung 7.2 und Abbildung 7.3 stellen den Kenntnisstand der Probanden grafisch dar.

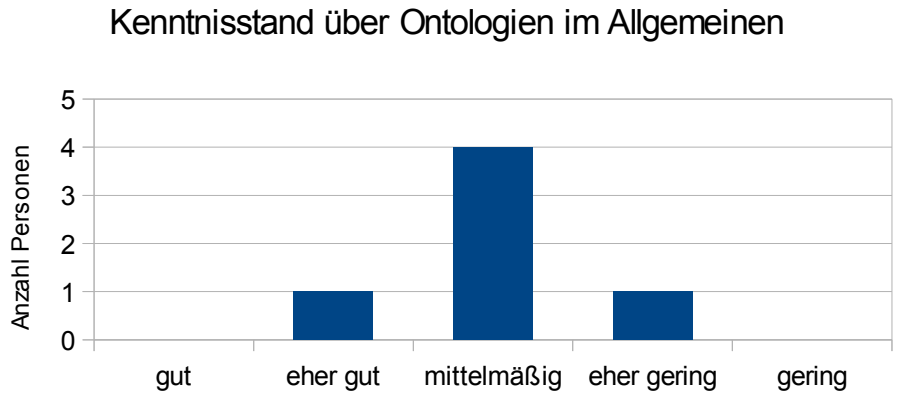


Abbildung 7.1.: Kenntnisstand über Ontologien im Allgemeinen.

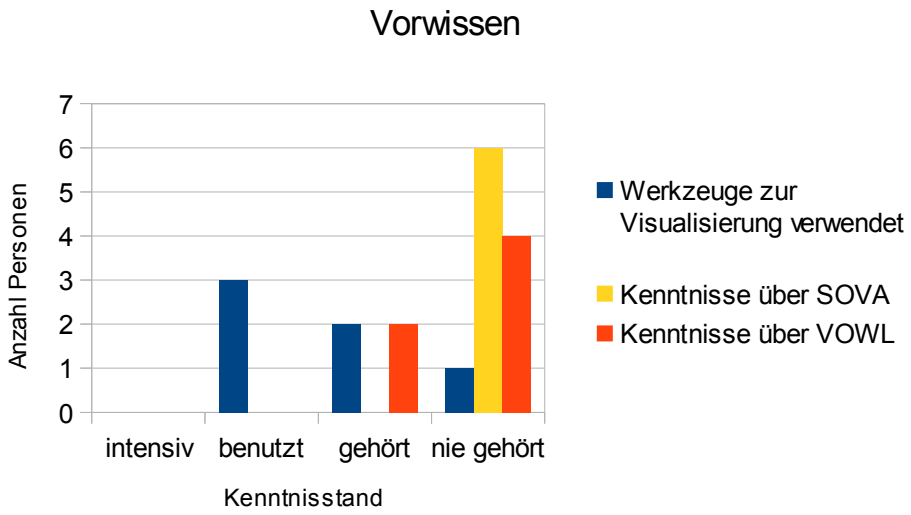


Abbildung 7.2.: Kenntnisstand über die Visualisierung von Ontologien und über die Visualisierungskonzepte SOVA und VOWL.

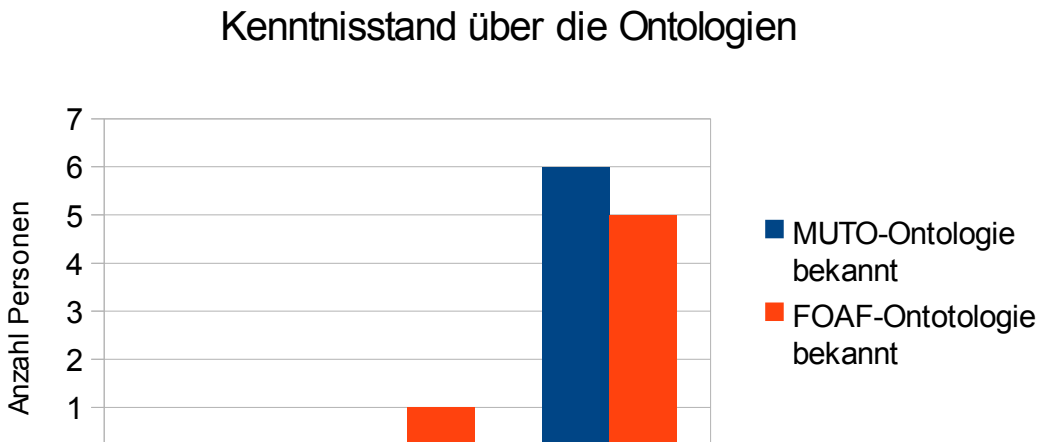


Abbildung 7.3.: Kenntnisstand über MUTO-Ontologie und die FOAF-Ontologie.

7.4. Resultate

In diesem Abschnitt werden die Ergebnisse der Studie sowie die Bewertungen der Visualisierungen durch die Probanden aufgelistet.
Die Studienteilnehmer konnten die Aufgaben 1, 2 und 4 in beiden Ontologien erfolgreich erledigen. Abbildung 7.4 verdeutlicht das Ergebnis für die MUTO-Ontologie und Abbildung 7.5

für die FOAF-Ontologie. Deutlich ist zu erkennen, dass Aufgabe 3 nicht von allen Probanden richtig beantwortet werden konnte.

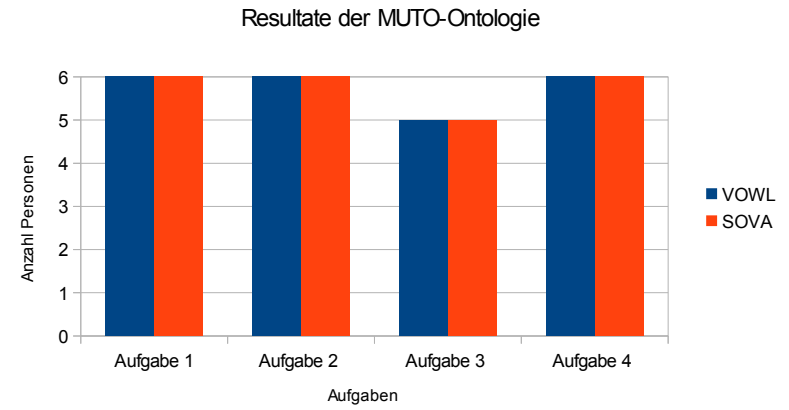


Abbildung 7.4.: Wieviele Personen konnten alle Fragen zur MUTO-Ontologie beantworten?

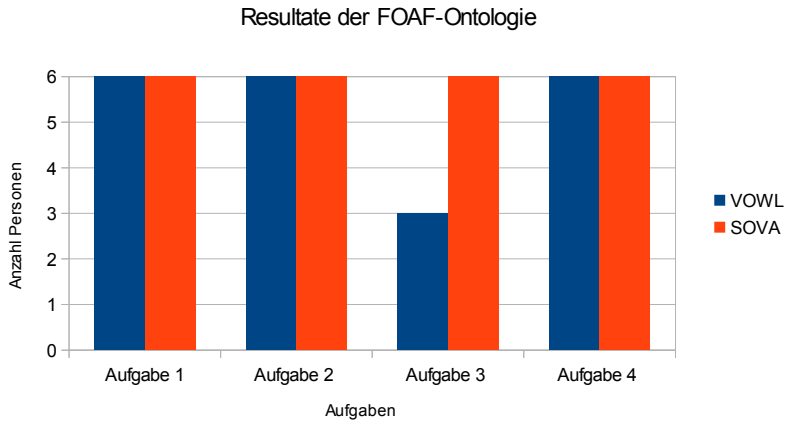


Abbildung 7.5.: Wieviele Personen konnten alle Fragen zur FOAF-Ontologie beantworten?

Abbildung 7.6 und Abbildung 7.7 erläutern die Ursache für das Abschneiden der Probanden bei der dritten Aufgabe. Während alle Teilnehmer Klassen und `Datatype Properties` innerhalb der VOWL-Visualisierung intuitiv erkennen konnten, schien dies auf `Object Properties` nicht zuzutreffen, ein einziger Studienteilnehmer beachtete `Object Properties` bei der Aufzählung aller Properties. Bei der SOVA-Visualisierung ergab sich ein ähnliches Bild. Wenige Teilnehmer konnten Klassen und `Object Properties` intuitiv unterscheiden. Berücksichtigt werden sollte allerdings, dass `Datatype Properties` von SOVA nicht dargestellt werden. Aus diesem Grund liegt die Wahrscheinlichkeit richtig zu raten bei 50 %. Fast allen Teilnehmern, denen eine intuitive Unterscheidung nicht gelang konnten durch logisches Denken eine richtige Differenzierung treffen und durch Berücksichtigung der Labels die Frage richtig beantworten.

Beispielsweise haben zwei von sechs Teilnehmern Properties in der SOVA-Visualisierung intuitiv erkannt. Alle vier Teilnehmern, denen dies nicht gelang, konnten durch logisches Schließen zu dem Schluss gelangen, dass Properties in SOVA durch die violette Farbe dargestellt werden. Dies erklärt das deutlich bessere Abschneiden der SOVA-Visualisierung bei der dritten Frage.

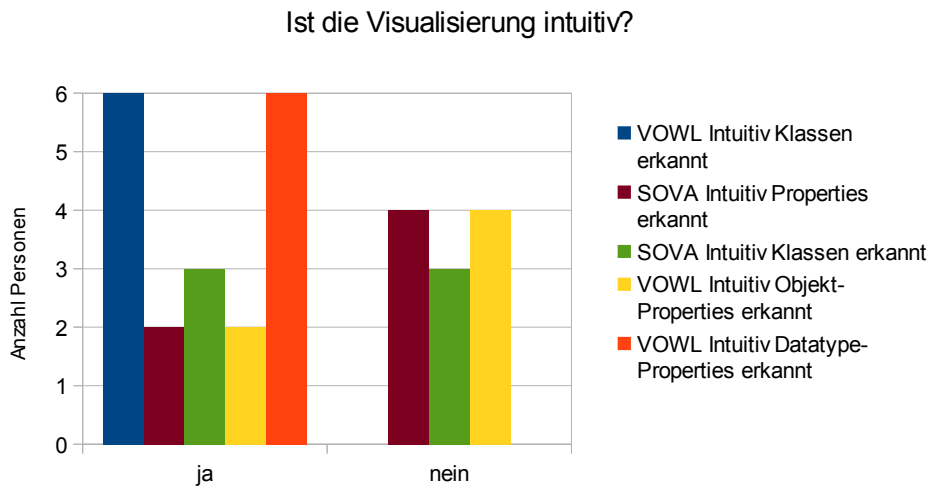


Abbildung 7.6.: Konnten die Probanden während der Evaluation Klassen und Properties intuitiv unterscheiden?

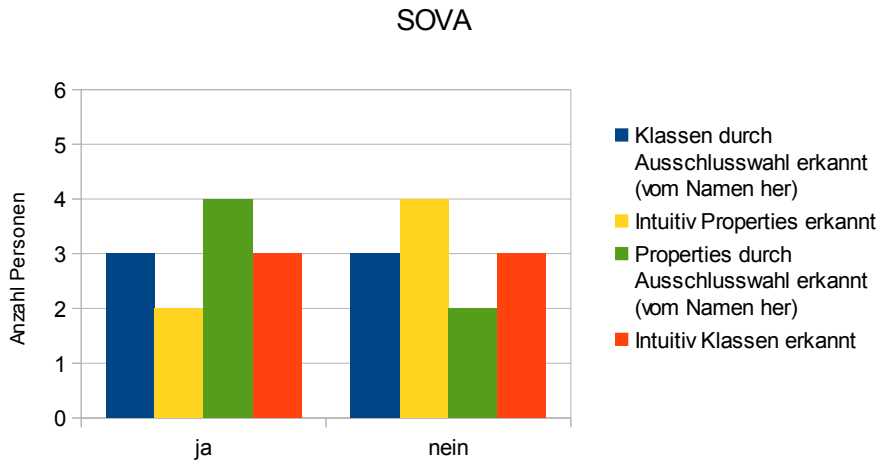


Abbildung 7.7.: Wie konnten die Probanden während der Evaluation Klassen und Properties unterschieden?

Alle sechs bevorzugten die VOWL-Visualisierung und benoteten VOWL sowohl bei der Gesamtbewertung als auch bei den Einzelbewertungen mit besseren Noten, als die Vergleichsvisualisierung. Die Bewertung erfolgte in Schulnoten, eine Eins gilt als beste, eine Sechs als schlechteste Note. Abbildung 7.8 vermittelt einen Eindruck der Gesamtbewertung durch die Probanden.

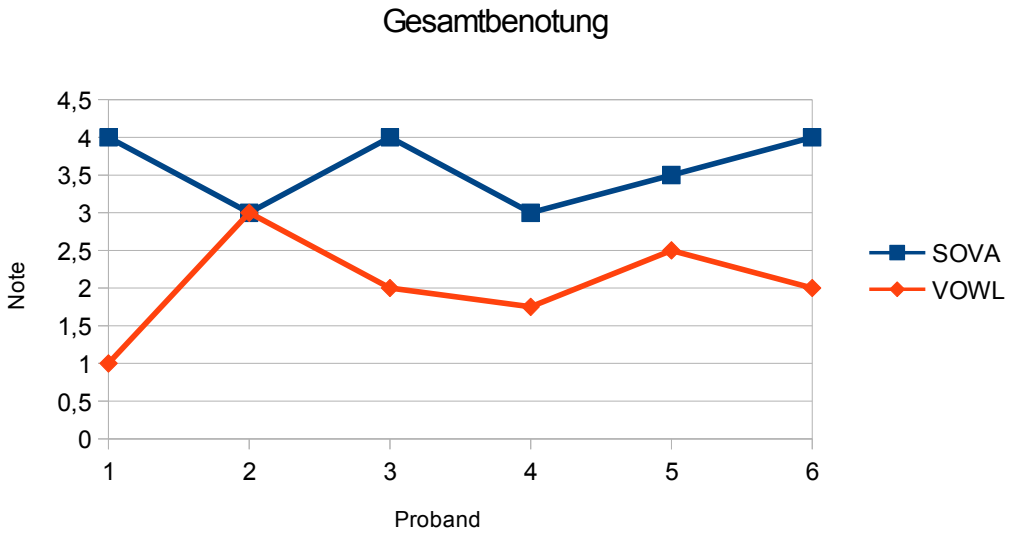


Abbildung 7.8.: Gesamtbenotung der Visualisierungskonzepte durch die Probanden.

Alle Probanden wählten zur Beantwortung von Frage 221 das im Zentrum platzierte Objekt „Person“. Durch das kräftebasierte Layout werden semantisch ähnlichere Objekte räumlich näher zueinander platziert, daher gruppieren sich jene Objekte mit der größten Ähnlichkeit im Zentrum der Ontologie. Während der Evaluation konnte man durch Verfolgen des Mauszeigers erkennen, dass dies ist, auch die Region, in der die meisten Menschen mit ihrer Suche starten.
Als Vorteil der SOVA-Visualisierung nannten fünf von sechs Probanden die Suchfunktion, während vier von sechs Probanden bei VOWL die Übersichtlichkeit sowie die deutlichen Unterschiede hinsichtlich Form und Farbe lobten. Während die Übersichtlichkeit bei VOWL gelobt wurde, wurde die Unübersichtlichkeit bei SOVA von vier von sechs Studienteilnehmern gerügt.
Bezüglich der bevorzugten Darstellung konnte keine Präferenz erkannt werden, drei Probanden bevorzugten die Darstellung aus VOWL 1.0, während drei Probanden die Optimierung bevorzugten. Beide Darstellungen konnten von 100 % aller Probanden beschrieben werden.

7.5. Fazit

Die Ergebnisse der Evaluation zeigen, dass sich mit der prototypischen Umsetzung aus Abschnitt 6 des in Kapitel 4 vorgestellten Lösungskonzepts Ontologien auch für Benutzer ohne Hintergrundwissen verständlich sind und sich ganzheitliche visualisieren lassen. Insbesondere Klassen und `Datatype Properties` lassen sich so intuitiv darstellen. Die Ergebnisse der Untersuchung zeigen, dass die Darstellung von `Object Properties` noch weitere optimiert werden kann und stellt damit ein Thema für eventuell auf diesem Konzept aufbauende Arbeiten dar. Die Resulte für die Visualisierung der `Datatype Properties` können aber auch durch die Fragestellung beeinflusst worden sein. Einigen Probanden der Studie war beispielsweise nicht klar, dass nach der Summe aller Properties gesucht wurde. In aufbauenden Arbeiten sollte dieser Sachverhalt daher noch einmal untersucht werden. Die Resultate zeigen ebenfalls die Eignung des Konzepts und dessen prototypische Umsetzung für größere Ontologien. Trotz sinkender Übersicht blieben die Auswirkungen auf die Probanden gering. Auch die Vorteile und Eignung eines kräftebasierten Layouts zur Visualisierung von Ontologien wurde durch die Studie gezeigt. Alle Studienteilnehmer bevorzugten die deutliche und klare Farbauswahl und Formwahl der VOWL-Visualisierung.

8. Zusammenfassung & Ausblick

In dieser Arbeit wurden existierende Visualisierungskonzepte zur grafischen Darstellung von Ontologien vorgestellt und auf ihre jeweiligen Stärken und Schwächen hin untersucht. Im Anschluss wurde das Visualisierungskonzept VOWL vorgestellt und optimiert, VOWL bildet dabei ein weiteres Konzept für die grafische Darstellung von Ontologien und versucht die zuvor genannten Schwächen zu vermeiden. Das VOWL-Visualisierungskonzept sollte auch für Benutzergruppen mit geringer bis keiner Erfahrung im Umgang mit Ontologien geeignet sein. Auf VOWL folgend wurde die Realisierung der Visualisierung beschrieben. Diese begann mit einer Untersuchung existierender Grafikframeworks, um herauszufinden, welche die Realisierung eines Prototypen positiv unterstützen könnten. Anschließend wurde das zeitliche Vorgehen der Realisierung und der Architektur der prototypischen Umsetzung als Protégé Plug-in beschrieben. Die Integration als Plug-in in ein weitverbreitetes Werkzeug zum Bearbeiten und Analysieren von Ontologien, wie Protégé, ermöglicht es sowohl fortgeschrittenen Anwender Protégé und Plug-ins in ihren Workflow zu integrieren, als auch Einsteigern, die auf Dokumentation und Tutorials rund um Protégé zurückgreifen können. Nach Abschluss der Realisierung wurde das optimierte Visualisierungskonzept und des darauf aufbauenden Prototyps, im Rahmen einer Nutzerstudie, evaluiert. Die Evaluation zeigt die Eignung von VOWL-Ontologien, auch für Nutzer mit wenigen Vorkenntnissen, kompakt und ganzheitlich darzustellen.

Ausblick

In diesem Abschnitt werden Optimierungsmöglichkeiten des Visualisierungskonzeptes und dessen prototypischer Umsetzung beschrieben.

Reaktion auf Änderungen in Protégé

Der Prototyp greift mittels OWL-API auf die von Protégé eingelesene Ontologie zu und speichert diese in einer eigenen Datenstruktur. Das Auslesen der durch Protégé eingelesenen Ontologie erfolgt einmalig bei der Initialisierung des Prototyps. Zum reinen Betrachten einer abgespeicherten Ontologie stellt dieser Sachverhalt keine Einschränkung dar, schließlich wird Protégé beim Laden einer abgespeicherten Ontologie ebenfalls neu initialisiert.
Mit Protégé hält der Benutzer ein mächtiges Werkzeug zum Bearbeiten und Erstellen von Ontologien in den Händen. Aufgrund des einmaligen Auslesens der Ontologie beim Initialisieren des Plug-ins ist der Prototyp in der aktuellen Ausbaustufe nicht in der Lage die Modifikationen des Benutzers innerhalb der Visualisierung darzustellen. Dasselbe gilt für

Ontologien, die zwar erstellt jedoch nicht gespeichert wurden.
Bisher müssen diese Ontologien zuerst gespeichert und anschließend geladen werden, damit sie von der prototypischen Umsetzung visualisiert werden können. Zur Lösung dieses Problems sind verschiedene Lösungsansätze denkbar. Beispielsweise könnte das Einlesen der abgespeicherten Ontologie auf den Moment verschoben werden, an dem das Plug-in durch den Benutzer aufgerufen wird. Alternativ könnte ein Update Mechanismus implementiert werden, der die Elemente des Prefuse-Datenmodells mit jenen des Protégé-Datenmodells vergleicht und Änderungen übernimmt. Aufgrund der eindeutigen Bezeichner wäre dieser Ansatz machbar.

Ausbau der Visualisierungskonzept für verschiedene Nutzergruppen

Das vorgestellte Visualisierungskonzept könnte verschiedene Nutzergruppen besser unterstützen. Denkbar wäre beispielsweise, dass die Symbolwahl von einer ausgewählten Nutzergruppe abhinge. Beispielsweise könnten für verschiedene Nutzergruppen unterschiedliche Symbole für `owl:intersectionOf` definiert werden. Zudem wäre es denkbar unerfahrenen Benutzern bestimmte Details vorzuenthalten, um ihnen die Nutzung und den Einstieg in das Plug-in zu erleichtern.

Ausbau der Visualisierungskonzept für sehr große Graphen

Das vorgestellte Visualisierungskonzept könnte für sehr große Graphen angepasst werden. Denkbar wäre es beispielsweise Informationen über Instanzen ab einer Mindestzoomstufe anzuzeigen. Dasselbe sei auch für verschiedene Property-Typen denkbar. Auf diese Weise könnte der Gesamtüberblick auf Kosten der Details verbessert werden. Die Details könnten erst angezeigt werden, falls der Benutzer einen Ausschnitt des Graphen vergrößere.
Zusätzlich könnte der Überblick bei großen Graphen durch eine zusätzliche Miniaturansicht weiter verbessert werden. Innerhalb der Evaluation aus Kapitel 7 schlug ein Teilnehmer Hinweise im Bereich der Darstellung vor, die ihn darüber informieren, dass außerhalb des sichtbaren Bereiches weitere Elemente existieren. Dies könnte über farbliche Markierungen, beispielsweise durch Pfeile, geschehen.

Konfigurierbarkeit der Visualisierung

Die prototypische Umsetzung könnte ausgebaut werden, um dem Nutzer mehr Möglichkeiten zu bieten, die Darstellung nach eigenen Wünschen zu verändern.

Vollständige Realisierung des Konzepts

Die prototypische Umsetzung könnte ausgebaut werden, um sämtliche Elemente des Visualisierungskonzeptes korrekt darzustellen. Kardinalitäten werden im Prototyp bisher nicht visuell dargestellt. Dasselbe gilt für die OWL-Elemente `FunctionalProperty`, `InverseFunctionalProperty` und `TransitiveProperty`.

Ausbau der unterstützen Formate

Der Prototyp liest die eingelesene Ontologie mittels OWL-API aus Protégé aus und kann daher prinzipiell alle Dateiformate visualisieren, die Protégé einlesen kann. Leider scheint die Unterstützung der verschiedener Dateiformate durch Protégé unterschiedlich ausgebaut zu sein. Falls eine Ontologie im RDF-Format eingelesen, durch Protégé in OWL-Format konvertiert und anschließend zurück in das RDF-Format exportiert wird, so konnte ein Datenverlust bemerkt werden. Getestet wurde die Funktionalität des Prototyps bisher nur mit Ontologien, die im RDF-Format vorlagen.

Unterstützung für Abfragesprachen

Die prototypische Umsetzung könnte erweitert werden, um Abfragesprachen direkt auf der visualisierten Ontologie ausführen zu können. Auf diese Weise könnten sehr erfahrene Nutzer eine visualisierte Ontologie besser explorieren.

Auflösung der Protégé Abhängigkeit

Der Prototyp könnte von seiner Abhängigkeit mit dem Protégé-Framework gelöst und als eigenständiges Programm veröffentlicht werden. Aus diese Weise wäre der Prototyp auch für Nutzer geeignet, die mit Protégé überfordert sind. Alternativ könnte der Prototyp als Plug-in für weitere Ontologie Editoren veröffentlicht werden, dies würde den Nutzerkreis weiter erhöhen.

A. Fragebogen

Identifikationsnummer:

Fragen zum Vorwissen:

Wie schätzen Sie Ihren Kentnissstand über **Ontologien im Allgemeinen** ab?	gut	eher gut	mittel-mäßig	eher gering	gering
	intensiv	schon benutzt	schon davon gehört	nie gehört	
Haben Sie bereits Werkzeuge zur **Visualisierung von Ontologien** verwendet?					
Kennen sie das Visualisierungskonzept **VOWL**?					
Kennen sie das Werkzeug / Plugin / Programm / Konzept: **SOVA**?					
	Ja	etwas	Nein	-	-
Kennen Sie die **MUTO** Ontologie? (Modular Unified Tagging Ontology)					
Kennen Sie die **FOAF** Ontologie? (Friend of a Friend)					

MUTO-Fragen

Plug-in **[ ] VOWL** **[ ] SOVA**

111. Bestimmen Sie den Namespace von “Item”.
112. Bestimmen Sie den Namespace von “tag of”.
113. Wieviele Klassen sind in dieser Ontologie enthalten?
114. Beschrieben Sie “Tagging”.
    Welche Aussagen können Sie über “Tagging” treffen?

Plug-in **[ ] VOWL** **[ ] SOVA**

121. Bestimmen Sie den Namespace von “Concept”.
122. Bestimmen Sie den Namespace von “nextTag”.
123. Wieviele Properties sind in dieser Ontologie enthalten?
124. Beschrieben Sie “Tag”.
    Welche Aussagen können Sie über “Tag” treffen?

FOAF-Fragen

Plug-in **[ ] VOWL** **[ ] SOVA**

211. Bestimmen Sie den Namespace von “Agent”.
212. Bestimmen Sie den Namespace von “yahooChatID”.
213. Wieviele Klassen sind in dieser Ontologie enthalten?
214. Beschrieben Sie “Concept”.
    Welche Aussagen können Sie über “Concept” treffen?

Plug-in **[ ] VOWL** **[ ] SOVA**

221. Bestimmen Sie den Namespace von “Person”.
222. Bestimmen Sie den Namespace von “birthday”.
223. Wieviele Properties sind in dieser Ontologie enthalten?
224. Beschrieben Sie “Document”.
    Welche Aussagen können sie über “Document” treffen?

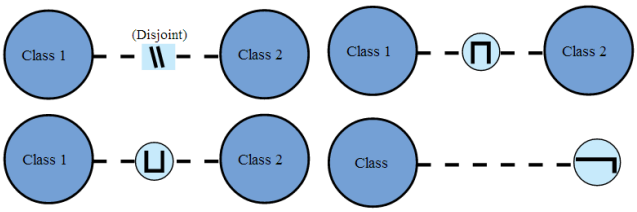
Fragen:

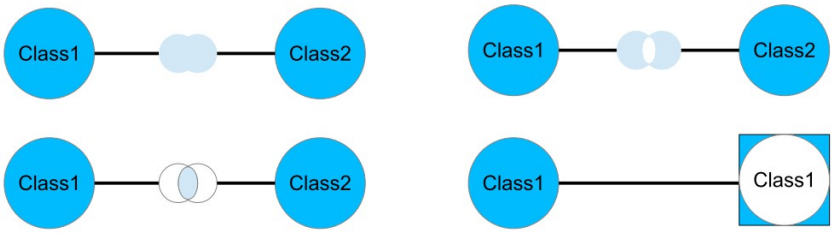
1. Was hat Ihnen gefallen an: OPTIONAL
   a. SOVA
   b. VOWL
2. Was hat Ihnen nicht gefallen an: OPTIONAL
   a. VOWL
   b. SOVA
3. Benoten Sie die Visualisierungen mit Schulnoten :
   (1 - sehr gut, 2 - gut, 3 - befriedigen, 4 - ausreichend, 5 - mangelhaft, 6 - ungenügend)
   a. SOVA
   b. VOWL
4. Wie schwer war die Bestimmung der Anzahl der Klassen bzw. Properties in Schulnoten:
   (1 - sehr gut, 2 - gut, 3 - befriedigen, 4 - ausreichend, 5 - mangelhaft, 6 - ungenügend)

	Klassen	Properties
a. VOWL		/
b. SOVA		/

5. Wie schwer war das Auslesen zusätzlicher Details in Schulnoten:
   (1 - sehr gut, 2 - gut, 3 - befriedigen, 4 - ausreichend, 5 - mangelhaft, 6 - ungenügend)
   a. SOVA
   b. VOWL
6. Welche Visualisierung würden Sie einsetzen und warum?

Zwei Darstellungen:





Fragen:

1. **Welche der beiden Darstellung gefällt Ihnen besser?**
   [ ] dunkelblaue / obere Darstellung
   [ ] hellblaue / untere Darstellung

2. **obere Darstellung**

   a. was bedeutet oben links?

   b. was bedeutet oben rechts?

   c. was bedeutet unten links?

   d. was bedeutet unten rechts?

3. **untere Darstellung**

   a. was bedeutet oben links?

   b. was bedeutet oben rechts?

   c. was bedeutet unten links?

   d. was bedeutet unten rechts?

B. Weitere Visualisierungen

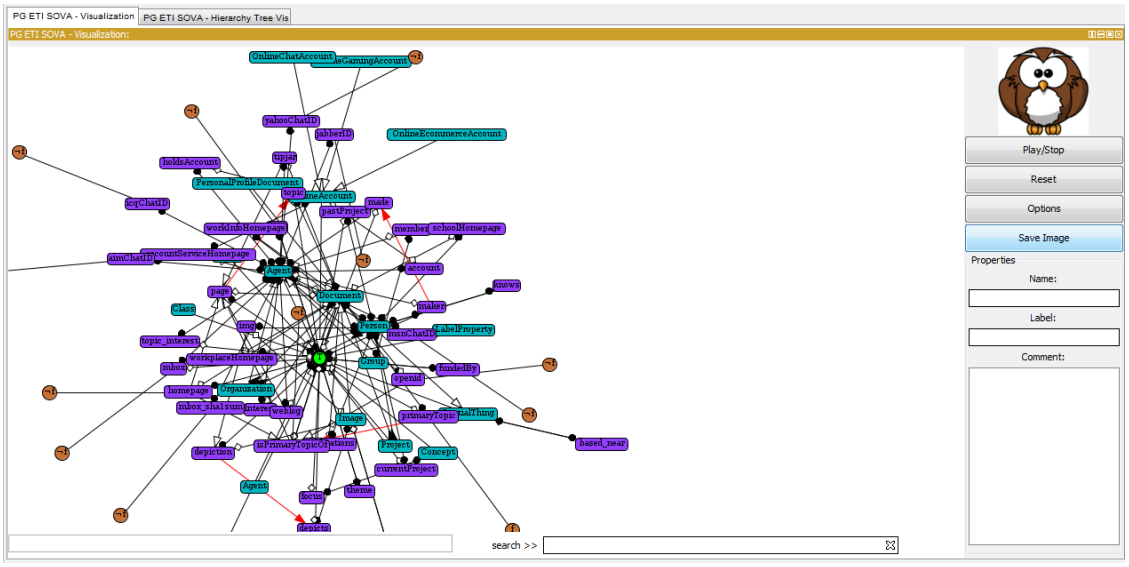


Abbildung B.1.: Visalisierung der FOAF-Ontologie mittels SOVA.

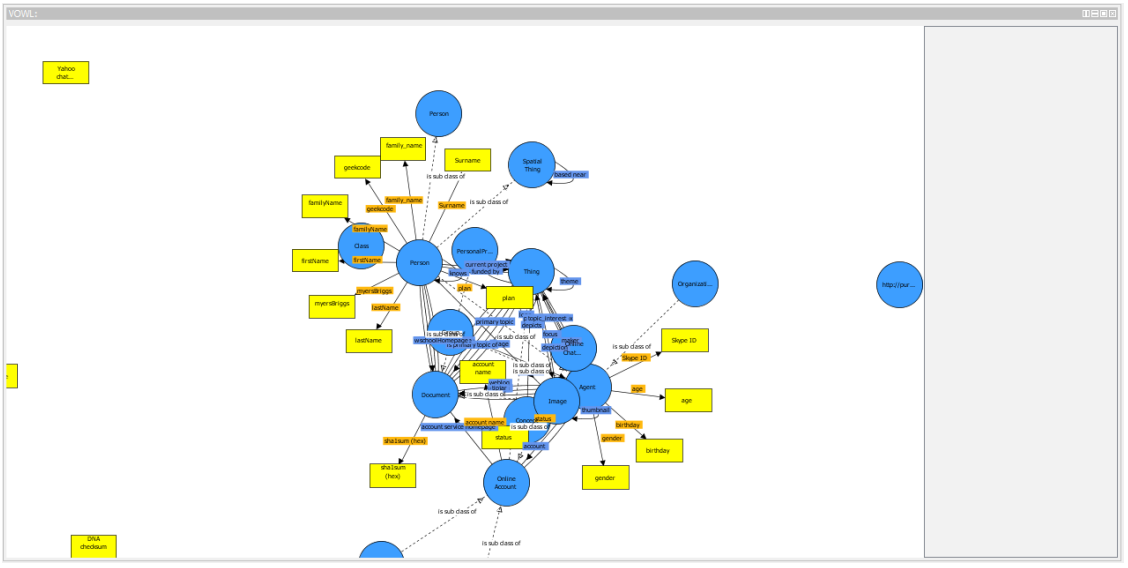


Abbildung B.2.: Visualisierung der FOAF-Ontologie mittels VOWL 2.0.

Literaturverzeichnis

[Ala03] H. Alani. TGVizTab: An Ontology Visualisation Extension for Protégé. In *Knowledge Capture (K-Cap'03), Workshop on Visualization Information in Knowledge Engineering*. 2003. Event Dates: October 26. (Zitiert auf den Seiten 9 und 45)

[Ald02] G. Alder. Design and implementation of the JGraph swing component. *Techical Report*, 1(6), 2002. (Zitiert auf Seite 64)

[Ald03] G. Alder. The JGraph Tutorial. *Veröffentlicht auf: http://www.jgraph.com/docs.html*, 37:62–71, 2003. (Zitiert auf Seite 64)

[Ber10] G. Bernstein. JUNG 2.0 Tutorial, 2010. URL http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf. (Zitiert auf Seite 66)

[Ber13] Berkeley Institute of Design (BiD). prefuse | interactive information visualization toolkit, 2013. URL http://prefuse.org. (Zitiert auf den Seiten 9, 29, 31, 69 und 70)

[BJKK10] T. Boiński, A. Jaworska, R. Kleczkowski, P. Kunowski. Ontology visualization. *Zeszyty Naukowe*, 18:15–20, 2010. (Zitiert auf Seite 70)

[BL89] T. Berners-Lee. Information Management: A Proposal. Technischer Bericht, CERN, 1989. URL http://www.w3.org/History/1989/proposal.html. (Zitiert auf Seite 13)

[BL94] T. Berners-Lee. Universal Resource Identifiers in WWW, 1994. URL http://tools.ietf.org/html/rfc1630. (Zitiert auf Seite 19)

[BL98] T. Berners-Lee. Semantic web road map, 1998. URL http://student.bus.olemiss.edu/files/conlon/others/others/semanticwebpapers/roadmap.pdf. (Zitiert auf Seite 13)

[BML97] N. S. Barghouti, J. M. Mocenigo, W. Lee. Grappa: a GRAPh PAckage in Java. In *In Fifth International Symposium on Graph Drawing*, S. 336–343. Springer-Verlag, 1997. (Zitiert auf den Seiten 10, 60, 61 und 62)

[BVHH+04] S. Bechhofer, F. Van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein, et al. OWL web ontology language reference. *W3C recommendation*, 10:2006–01, 2004. (Zitiert auf den Seiten 24 und 25)

[CT98] I. F. Cruz, R. Tamassia. Graph drawing tutorial. *URL: http://www4.ncsu.edu/~gremaud/MA432/graph_drawing_tutorial.pdf*, 1998. (Zitiert auf Seite 27)

[DB10] L. M. Dan Brickley. FOAF Vocabulary Specification 0.98, 2010. URL http://xmlns.com/foaf/spec/. (Zitiert auf Seite 100)

[G+93] T. R. Gruber, et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993. (Zitiert auf Seite 15)

[Gan13] E. R. Gansner. Using Graphvizas a Library (cgraph version). 2013. (Zitiert auf Seite 60)

[GR11] J. Gantz, D. Reinsel. Extracting value from chaos. *IDC iView*, S. 1–12, 2011. (Zitiert auf Seite 13)

[Gra10] GraphStream - Gallery, 2010. URL http://graphstream-project.org/doc/Gallery. (Zitiert auf den Seiten 10, 75 und 76)

[Gra13a] Graphviz - Graph Visualization Software, 2013. URL http://www.graphviz.org/Home.php. (Zitiert auf den Seiten 58 und 59)

[Gra13b] Node Shapes, 2013. URL http://www.graphviz.org/doc/info/shapes.html. (Zitiert auf Seite 60)

[HB11] M. Horridge, S. Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011. (Zitiert auf Seite 24)

[HCL05] J. Heer, S. K. Card, J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, S. 421–430. ACM, 2005. (Zitiert auf den Seiten 69 und 70)

[Hes02] W. Hesse. Ontologie (n). *Informatik-Spektrum*, 25(6):477–480, 2002. (Zitiert auf Seite 15)

[Hit07] P. Hitzler. *Semantic Web: Grundlagen*. Springer, 2007. URL http://books.google.de/books?id=lwa7TuJ_RR8C. (Zitiert auf den Seiten 15, 20 und 21)

[Hor10] M. Horridge. OWLViz, 2010. URL http://protegewiki.stanford.edu/wiki/OWLViz. (Zitiert auf Seite 59)

[Ins13a] Institut für Medizinische Informatik an der Universität Stanford (USA). The Protégé Ontology Editor and Knowledge Acquisition System, 2013. URL http://protege.stanford.edu. (Zitiert auf den Seiten 24 und 26)

[Ins13b] Institut für Medizinische Informatik an der Universität Stanford (USA). The Protégé Ontology Editor and Knowledge Acquisition System, 2013. URL http://protege.stanford.edu/download/registered.html. (Zitiert auf den Seiten 24 und 25)

[Ins13c] Institut für Medizinische Informatik an der Universität Stanford (USA). protégé-owl api, 2013. URL http://protege.stanford.edu/plugins/owl/api/index.html. (Zitiert auf Seite 26)

[Ins13d] Institut für Medizinische Informatik an der Universität Stanford (USA). what is protégé-frames?, 2013. URL http://protege.stanford.edu/overview/protege-frames.html. (Zitiert auf Seite 25)

[Ins13e] Institut für Medizinische Informatik an der Universität Stanford (USA). what is protégé-owl?, 2013. URL http://protege.stanford.edu/overview/protege-owl.html. (Zitiert auf Seite 25)

[Int13] International Semantic Web Conference. Keynote - Ramanathan V. Guha, 2013. URL http://iswc2013.semanticweb.org/content/keynote-ramanathan-v-guha. (Zitiert auf Seite 18)

[Jun] (Zitiert auf den Seiten 10, 66 und 68)

[Jun10a] ProjectsUsingJUNG, 2010. URL http://sourceforge.net/apps/trac/jung/wiki/ProjectsUsingJUNG. (Zitiert auf Seite 66)

[JUN10b] JUNG Framework Development Team. JUNG Java Universal Network/Graph Framework, 2010. URL http://jung.sourceforge.net/. (Zitiert auf Seite 66)

[KN+91] E. Koutsofios, S. North, et al. Drawing graphs with dot. Technischer Bericht, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991. (Zitiert auf den Seiten 10, 12, 59 und 61)

[KTH+06] A. Katifori, E. Torou, C. Halatsis, G. Lepouras, C. Vassilakis. A Comparative Study of Four Ontology Visualization Techniques in Protege: Experiment Setup and Preliminary Results. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, S. 417–423. 2006. doi:10.1109/IV.2006.3. (Zitiert auf den Seiten 45 und 46)

[KV06] A. Kirpal, A. Vogel. Neue Medien in einer vernetzten Gesellschaft: Zur Geschichte des Internets und des World Wide Web. *NTM International Journal of History & Ethics of Natural Sciences, Technology & Medicine*, 14(3):137–147, 2006. (Zitiert auf Seite 13)

[KWV07] S. Krivov, R. Williams, F. Villa. GrOWL: A tool for visualization and editing of OWL ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):54–57, 2007. (Zitiert auf den Seiten 9, 33 und 35)

[LDA11] S. Lohmann, P. Díaz, I. Aedo. MUTO: the modular unified tagging ontology. In *Proceedings of the 7th International Conference on Semantic Systems*, I-Semantics '11, S. 95–104. ACM, New York, NY, USA, 2011. doi:10.1145/2063518.2063531. URL http://doi.acm.org/10.1145/2063518.2063531. (Zitiert auf den Seiten 9, 12, 22, 23, 42, 81 und 100)

[Lei13] H. Leitte. Darstellung von Graphen, 2013. URL http://www.iwr.uni-heidelberg.de/groups/CoVis/Data/vis1-9_Graphen.pdf. (Zitiert auf Seite 26)

[Luc13] H.-D. Luckhardt. Informationsvisualisierung, 2013. URL http://wiki.infowiss.net/Informationsvisualisierung. Revision vom 19. April 2011 geändert durch Heinz-Dirk Luckhardt, URL http://wiki.infowiss.net/index.php?title=Informationsvisualisierung&oldid=14669. (Zitiert auf Seite 27)

[MGH+09] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz. Owl 2 web ontology language: Profiles. *W3C recommendation*, 27:61, 2009. (Zitiert auf Seite 24)

[Min12] Miniwatts Marketing Group. Internet World Stats, 2012. URL http://www.internetworldstats.com/stats.htm. (Zitiert auf Seite 13)

[Moz13] Mozilla Foundation. Mozilla Public License, 2013. URL http://www.mozilla.org/MPL. Letzte Änderung vom 1. Februar 2012. (Zitiert auf Seite 24)

[NL13] S. Negru, S. Lohmann. A Visual Notation for the Integrated Representation of OWL Ontologies. In *Proceedings of the 9th International Conference on Web Information Systems and Technologies*, WEBIST '13, S. 308–315. SciTePress, 2013. (Zitiert auf den Seiten 10, 11, 50, 51, 52 und 94)

[Nor04] S. C. North. Drawing graphs with NEATO. *NEATO User Manual*, S. 11, 2004. (Zitiert auf den Seiten 10, 59 und 60)

[PK10] T. B. Piotr Kunowski. SOVA - Visualization symbols, 2010. URL http://protegewiki.stanford.edu/images/1/12/SOVA-Symbols.pdf. (Zitiert auf den Seiten 9, 38, 39, 40 und 41)

[PK12] T. B. Piotr Kunowski. SOVA, 2012. URL http://protegewiki.stanford.edu/wiki/SOVA. (Zitiert auf den Seiten 9, 18 und 38)

[Pro13a] Protege Client-Server, 2013. URL http://protegewiki.stanford.edu/wiki/Protege_Client-Server. Revision vom 27. April 2011, URL http://protegewiki.stanford.edu/index.php?title=Protege_Client-Server&oldid=9805. (Zitiert auf Seite 26)

[Pro13b] Protege-Frames, 2013. URL http://protegewiki.stanford.edu/wiki/Protege-Frames. Revision vom 16. März 2012, URL http://protegewiki.stanford.edu/index.php?title=Protege-Frames&oldid=10874. (Zitiert auf Seite 26)

[Pro13c] Protege-OWL, 2013. URL http://protegewiki.stanford.edu/wiki/Protege-OWL. Revision vom 16. März 2012, URL http://protegewiki.stanford.edu/index.php?title=Protege-OWL&oldid=10879. (Zitiert auf Seite 26)

[SA11] R. Sivakumar, P. Arivoli. ONTOLOGY VISUALIZATION PROTÉGÉ TOOLS–AReview. *International Journal of Advanced Information Technology*, 1(4), 2011. (Zitiert auf Seite 45)

[Sch11] H. Schmidt. Deutschlands Internet-Industrie ist schlecht gerüste, 2011. URL http://blogs.faz.net/netzwirtschaft-blog/2011/05/27/deutschlands-internet-industrie-ist-schlecht-geruestet-2574/. (Zitiert auf Seite 13)

[Sch13] G. Scheuermann. Darstellung von Graphen, 2013. URL http://www.informatik.uni-leipzig.de/bsv/homepage/sites/default/files/Infovis_5-graphs_0.pdf. (Zitiert auf Seite 30)

[Sim96] S. Sim. Automatic graph drawing algorithms. *Manuscript, available at http://www.drsusansim.org/papers/grafdraw.pdf*, 1996. (Zitiert auf Seite 27)

[Ste13] F. Steeg. Zest/DOT, 2013. URL http://wiki.eclipse.org/Zest/DOT. Revision vom 21. August 2013, URL http://wiki.eclipse.org/index.php?title=Zest/DOT&oldid=346056. (Zitiert auf den Seiten 12 und 63)

[Ten13] Tensegrity Software GmbH. Layout Algorithmen, 2013. URL http://www.tensegrity-software.de/layout-algorithmen/html. (Zitiert auf den Seiten 29 und 30)

[TM06] I. R. Tamara Mchedlidze, Martin Nöllenburg. Algorithmen zur Visualisierung von Graphen, 2006. URL http://i11www.iti.uni-karlsruhe.de/_media/teaching/winter2012/graphdrawing/v15.pdf. (Zitiert auf Seite 30)

[Ver13] Verkehrs- und Tarifverbund Stuttgart GmbH (VVS). Liniennetz, 2013. URL http://www.vvs.de/karten-plaene/liniennetz. Verbund-Schienennetz (pdf). (Zitiert auf den Seiten 9 und 28)

[Vog11] L. Vogel. Eclipse Zest - Tutorial, 2011. URL http://www.vogella.com/articles/EclipseZest/article.html. (Zitiert auf den Seiten 10, 12 und 63)

[VVVM04] T. Version, L. Version, P. Version, B. McBride. RDF Vocabulary Description Language 1.0: RDF Schema. *Changes*, 2004. (Zitiert auf den Seiten 19 und 25)

[Wik13] Hypertext - Wikipedia, 2013. URL http://de.wikipedia.org/wiki/Hypertext. Revision vom 08. August 2013, URL http://de.wikipedia.org/w/index.php?title=Hypertext&oldid=121327895. (Zitiert auf Seite 13)

Alle URLs wurden zuletzt am 12. Dez. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift