Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 210

Entwicklung eines Benchmark-Generators zum Testen von Ontologievisualisierungen

Vincent Link

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	Dr. Steffen Lohmann, Dipl.-Inf. Florian Haag
Beginn am:	15. April 2015
Beendet am:	15. Oktober 2015
CR-Nummer:	I.2.4, I.7.2

Kurzfassung

Mit Ontologien wird die Struktur von Informationen und Wissen hauptsächlich in der Web Ontology Language (OWL) beschrieben. Aufgrund des Umfangs und der hohen Komplexität solcher Ontologien werden mit Ontologievisualisierungen die beschriebenen Konzepte und deren Beziehungen anschaulicher dargestellt. Damit diese Visualisierungen effektiv getestet werden können, werden Benchmark-Ontologien benötigt. In dieser Arbeit wird *OntoBench*, ein Generator für solche Ontologien, in Form einer Webanwendung konzipiert und implementiert, sodass sich diese systematisch und flexibel erstellen lassen. Mit den generierten Ontologien soll primär die Konformität zur OWL-Spezifikation testbar sein; die Skalierbarkeit oder Performanz der Visualisierung stehen hierbei nicht im Vordergrund. Abschließend wird dieser Ansatz an drei Visualisierungen evaluiert und ein Ausblick auf mögliche Erweiterungen gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

Im Zuge des Semantic Webs und seiner steigenden Verbreitung müssen Wissen und Informationen strukturiert zugänglich gemacht werden. Mit Hilfe von Ontologien werden diese Strukturen beschrieben, sodass sie einheitlich vorliegen und auch von Computern interpretiert werden können. Die Ontologien sind größtenteils in der *Web Ontology Language* (OWL) geschrieben und aufgrund des großen Umfangs und der hohen Komplexität nur schwer vom Menschen erfassbar. Um bei dieser Aufgabe zu unterstützen, entstanden eine Reihe von Ontologievisualisierungen, die die beschriebenen Konzepte und Beziehungen anschaulich darstellen.

Wird eine Visualisierung für OWL-Ontologien entwickelt, sollte sie mittels einer geeigneten Ontologie auf ihre korrekte Funktionalität getestet werden. Reale Ontologien sind hier nicht zweckmäßig, da sie nicht alle möglichen OWL-Konstrukte verwenden, und geeignete Benchmark-Ontologien sind rar. Es ist durchaus möglich, dass eine Ontologievisualisierung auch ohne Benchmark-Ontologien fehlerfrei entwickelt sein kann; bisher gibt es jedoch kaum Möglichkeiten, wie diese Eigenschaft bestätigt werden könnte.

In dieser Arbeit wird die Anwendung *OntoBench* konzipiert und implementiert, mit der sich Benchmark-Ontologien hierfür systematisch und flexibel erstellen lassen. Mit generierten Ontologien wird der Ansatz gegen Ende an drei Visualisierungen getestet. Abschließend wird ein Fazit gezogen und auf mögliche Erweiterungen und Forschungsbereiche eingegangen.

Gliederung

Der Inhalt dieser Arbeit ist über die weiteren Kapitel wie folgt strukturiert:

Kapitel 2 – Grundlagen und verwandte Arbeiten: Im nächsten Kapitel werden kurz und bündig die fachlichen Grundlagen dieser Arbeit nahegebracht. Anschließend werden verwandte Arbeiten betrachtet und diese Arbeit zwischen diesen positioniert.

Kapitel 3 – Konzept: Alle Entwurfs- und Designentscheidungen bezüglich des Generators und der generierten Ontologien werden in diesem Kapitel begründet. Damit stellt es die Grundlage für die Implementierung dar.

Kapitel 4 – Implementierung: In diesem Kapitel wird auf die technischen Details der Implementierung eingegangen. Nach Skizzieren der Architektur werden die einzelnen Komponenten der Anwendung beschrieben.

Kapitel 5 – Evaluation: Der für OntoBench gewählte Ansatz wird im vorletzten Kapitel zum einen anhand der Anwendung selbst und außerdem auch an drei Visualisierungen evaluiert.

Kapitel 6 – Zusammenfassung und Ausblick: Mit einem Fazit und einem Ausblick auf mögliche Erweiterungen, wird die Arbeit in diesem Kapitel abschlossen.

2. Grundlagen und verwandte Arbeiten

Im Folgenden wird zuerst eine kurze Übersicht über die fachlichen Themen gegeben. Anschließend werden verwandte Arbeiten nach bestem Wissen beschrieben und mit dieser Arbeit verglichen.

2.1. Fachliche Grundlagen

Bevor direkt in die fachliche Diskussion eingestiegen wird, soll eine kurze Einführung in das Themengebiet gegeben werden.

2.1.1. Semantic Web

Im heutigen Internet werden Informationen hauptsächlich in einer für den Menschen lesbaren Form angeboten. Von Computern können diese schlecht verarbeitet werden, da ihre Form zum einen nicht standardisiert ist und zum anderen der Inhalt aus Texten nur schwer extrahiert werden kann. Das Semantic Web soll deshalb dem heutigen Internet eine Struktur geben, welche die Informationen in einer für Computer interpretierbaren Form im sogenannten *Web of data* beschreibt. [BLHL+01]

2.1.2. Ontologien

Über das Semantic Web allein erhalten Computer nur den reinen Zugriff auf Informationen. Damit diese aber genutzt werden können, wird in Ontologien zum einen die Struktur der Informationen beschrieben. Computer können dadurch unterschiedliche Informationen miteinander verknüpfen und interpretieren. Zum anderen enthalten Ontologien Inferenzregeln, mit denen Rückschlüsse aus in dieser Struktur vorliegenden Daten gezogen werden können. [BLHL+01]

In einem vereinfachten Beispiel liegen dem Computer *Vorlesungen* und *Studenten* als Daten vor, über die sonst keine Informationen bekannt sind. Sagt man mit einer Ontologie, dass Studenten Vorlesungen besuchen können, kann der Computer diese nun verknüpfen. Erweitert man die Ontologie um die Information, dass ein Student immatrikuliert sein muss, um eine Vorlesung zu besuchen, können alle Studenten gefunden werden, die keine Vorlesung besuchen dürfen. Zusammengefasst, kann der Computer durch eine Ontologie Schlussfolgerungen aus den Daten ziehen.

2.1.3. OWL

Damit Ontologien von einem Computer interpretiert werden können, müssen sie in einer Sprache formuliert sein, die er verstehen kann. Eine solche Sprache ist die *Web Ontology Language*, kurz *OWL*, welche die Empfehlung des *World-Wide-Web-Consortium* ist und deshalb als einzige in dieser Arbeit berücksichtigt wird. OWL 2 ist die aktuelle Version der Web Ontology Language, welche OWL 1 um neue Konstrukte erweitert. [W3C12]

Ein Großteil von OWL lässt sich durch die drei Grundelemente *Classes*, *Properties* und *Individuals*, auf Deutsch *Klassen*, *Eigenschaften* und *Individuen*, erklären. Eine Klasse ist eine Abstraktion von konkreten Dingen und gruppiert diese anhand gemeinsamer Merkmale. Solch ein konkretes Ding wird als Individuum bezeichnet. Individuen einer Klasse können in Relation mit Individuen anderer oder derselben Klasse stehen, was als Property bezeichnet wird. [HKP+12]

Auf das obige Beispiel bezogen, stellen *Vorlesung* und *Student* Klassen dar. Die Klasse Student hat dabei die Eigenschaft, dass sie die Klasse Vorlesung besucht. Das Individuum *Höhere Mathematik* könnte also von dem Individuum *Barbara Mustermann* besucht werden.

Klassen beziehungsweise Individuen können zwei unterschiedliche Arten von Eigenschaften verbinden. Eine Vorlesung hat beispielsweise die Eigenschaft *maximale Teilnehmeranzahl*. Hierbei spricht man von einer *Data Property*, da sie einen reinen Wert beschreibt. Mit einer *Object Property* werden im Gegensatz dazu Eigenschaften beschrieben, die sich auf eine Klasse beziehungsweise deren Individuen beziehen.

OWL definiert mehrere Sprachkonstrukte, mit denen unter anderem diese drei Grundelemente beschrieben werden. Klassen lassen sich zum Beispiel durch eine explizite Aufzählung der Individuen oder auch durch eine Vereinigung anderer Klassen definieren. [BHH+04]

2.1.4. OWL-Profile

OWL-Profile definieren Subsets der Konstrukte von OWL, welche für unterschiedliche Anwendungsgebiete von Ontologien benötigt werden. Eine Ontologie ist mit einem bestimmten Profil konform, wenn sie nur die Konstrukte verwendet, die in diesem definiert sind.

So sollten manche OWL-Konstrukte nicht in einer Ontologie verwendet werden, um beispielsweise beim *Reasoning* möglichst effizient Schlussfolgerungen aus einer Ontologie und Daten ziehen zu können. [BHH+04]

2.2. OntoViBe

Haag et al. stellen mit OntoViBe [HLNE14a] ein Set von Ontologien zum Testen von Ontologievisualisierungen vor. Momentan befindet sich OntoViBe in Version 2, welche den in [HLNE14b] vorgestellten Vorgänger um weitere Testfälle und Elemente von OWL erweitert.

Die Ontologien decken einen großen Bereich der OWL 2-Konstrukte ab. Sie enthalten außerdem auch *DC*-Annotationen [DCM12] und Konstrukte, die für Testfälle der hauptsächlich graphbasierten

Ontologievisualisierungen gedacht sind. In einem dieser Testfälle verbinden zum Beispiel mehrere Properties zwei Klassen, wodurch in einem Graphen die Überlagerung von Kanten getestet werden kann.

Beim Design wurde darauf geachtet, dass die einzelnen Ontologien möglichst kompakt sind und in der Visualisierung somit wenige Elemente wiederholt werden. Das wird zum einen dadurch erreicht, indem bereits definierte Elemente eines Typs von anderen Elementen wiederverwendet werden, wodurch sich eine starke Verflechtung der einzelnen Testfälle ergibt. Zum anderen erzeugt die Aufteilung auf vier Ontologien in geringem Maße eine Modularität, sodass nicht alle Testfälle auf einmal visualisiert werden müssen.

Mit der *Minimal*-Ontologie bietet OntoViBe eine *leere* Ontologie, die für eine Visualisierung das Minimum einer gültigen Eingabemenge darstellt. Die *Core*-Ontologie enthält die meisten Elemente von OntoViBe und deckt einen großen Bereich von OWL 2 ab. Sie referenziert Elemente aus der *Imported*-Ontologie, um den Fall einer Visualisierung von mehreren zusammenhängenden Ontologien zu testen. Aufgrund des großen Spielraums an möglichen Kombinationen von Kardinalitäten, liegen diese separat in der *Cardinality*-Ontologie.

2.3. Lehigh University Benchmark und Erweiterungen

In [GPH05] stellen Guo et al. mit dem *Lehigh University Benchmark* (LUBM) eine Test-Suite für Ontologiesysteme vor. In diesen Systemen können Daten aus dem Semantic Web gespeichert, abgefragt oder für das Reasoning werden. Die Suite baut auf Benchmarks für Datenbanken auf, welche aber an die Eigenschaften des Semantic Webs angepasst sind.

LUBM besteht aus mehreren Komponenten, von denen die Ontologie nur einen Teil darstellt. Mit einem Generator für Instanzen von Klassen der Ontologie, können beliebig große Testdaten generiert werden, um sowohl kleinere als auch große Systeme zu testen. Mit einem Set von Abfragen kann das System anschließend unter Verwendung von vordefinierten Performanzmetriken bewertet werden.

Die Ontologie selbst basiert auf Benchmark-Daten, die die Struktur des Universitätskontexts beschreibt. Sie verwendet außerdem nur die Syntax des OWL Lite-Profils, mit der ein effizientes Reasoning in dem getesteten System genutzt werden kann.

2.3.1. Lehigh Bibtex Benchmark

Der in [WGQH05] vorgestellte *Lehigh Bibtex Benchmark* verbessert den Generator der LUBM-Suite. Mit Hilfe eines probabilistischen Modells können im Lehigh Bibtex Benchmark auch Instanzen von Klassen beliebiger Ontologien generiert werden, welche sich also nicht unbedingt auf den Universitätskontext beziehen müssen. Dadurch kann mit dieser Benchmark-Suite ein breiteres Spektrum an Anwendungsgebieten modelliert und getestet werden.

2.3.2. University Ontology Benchmark

Ma et al. erweitern mit dem *University Ontology Benchmark* (UOBM) die LUBM-Suite, um das Testen von Reasonern und der Skalierbarkeit eines mit Ontologien arbeitenden Systems zu verbessern.

UOBM bietet zwei Ontologien, die die OWL Lite-Syntax beziehungsweise die OWL DL-Syntax im Gegensatz zu LUBM vollständig abdecken. Außerdem wurde der Generator in der Hinsicht optimiert, dass er die generierten Instanzen besser miteinander verknüpft, woraus sich insgesamt realitätsnahe Daten ergeben. [MYQ+06]

2.4. W3C-Testontologien

Vom *World-Wide-Web-Consortium* wurden mit den Spezifikationen von OWL 1 [CR04] und OWL 2 [SHKG12] auch verschiedene, kleinere Testontologien veröffentlicht. Mit diesen Ontologien soll zum einen die korrekte Verwendung von OWL gezeigt werden. Zum anderen sind sie aber auch dafür gedacht, dass Anwendungen, die mit OWL-Ontologien arbeiten, auf ihre Konformität zur Spezifikation getestet werden können.

Der größte Teil der Ontologien ist zum Testen von Syntax, speziellen Problemstellungen von OWL oder Reasonern gedacht. Für wenige OWL-Konstrukte existieren jedoch auch eigene Testontologien, sodass diese separat getestet werden können. Bei einem großen Teil der Ontologien sind außerdem auch die OWL-Profile angegeben, zu denen die jeweilige Ontologie konform ist.

2.5. Protégé und andere Ontologie-Editors

Protégé[1] ist ein Ontologie-Editor, der ein mächtiges Framework und Vielzahl darauf aufbauender Plugins, wie unter anderem Visualisierungen, mit sich bringt. Knublauch et al. bezeichnen Protégé in [KHM+05] als die führende Anwendung zum Erstellen und Bearbeiten von Ontologien. Aufgrund seiner weiten Verbreitung und der guten Unterstützung von OWL, wähle ich Protégé repräsentativ für andere, ähnliche Ontologie-Editors.

Mit einem Ontologie-Editor könnten eigene Benchmark-Ontologien zum Testen einer Visualisierung erstellt werden. Protégé bietet dabei einen großen Spielraum an Kombinationsmöglichkeiten der OWL-Konstrukte, womit eine Ontologie auf eine bestimmte Ontologievisualisierung zugeschnitten werden kann. Für den Verlauf der Entwicklung einer Visualisierung kann sich diese Methode eignen, da einzelne, inkrementelle Anpassungen an der Benchmark-Ontologie mit wenig Aufwand vorgenommen werden können.

Wird solch eine Ontologie mehrmals in unterschiedlicher Form benötigt, lohnt sich der Aufwand aufgrund der großen Änderungen nicht. So sind mehrere Benutzereingaben nötig, um bereits kleinere Testfälle zu konstruieren. Neben der Tatsache, dass zu Beginn geeignete Testfälle gewählt werden

[1] http://protege.stanford.edu/

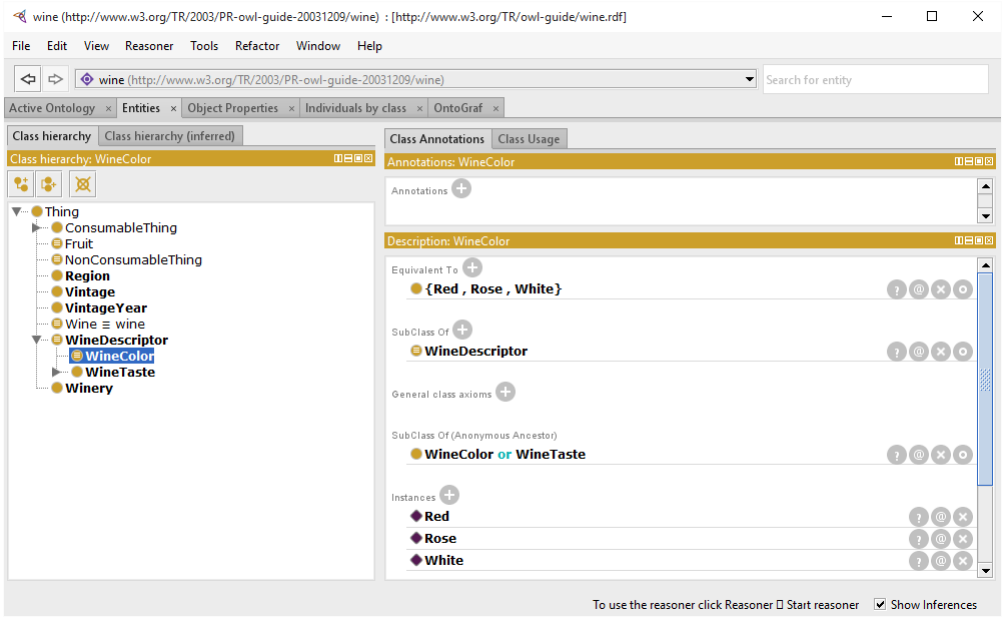

Abbildung 2.1.: Die Oberfläche des Ontologie-Editors Protégé, in dem die Wine-Ontologie bearbeitet wird.

müssen, erfordert die Bedienung eines Ontologie-Editors unter Umständen eine gute Kenntnis des Programms, um überhaupt alle OWL-Konstrukte finden zu können.

2.6. Überblick und Zusammenfassung

Bei den meisten der aufgeführten Arbeiten liegt der Fokus auf einem anderen Thema als bei dieser Arbeit. Sie sind zum Testen von Reasonern und Ontologiesystemen gedacht, und versuchen diese durch große Datenmengen oder auch spezielle OWL-Konstrukte zu vergleichen. Außerdem sind die Ontologien dieser Arbeiten auf OWL 1 beschränkt und unterstützen Konstrukte von OWL 2 somit nicht.

OntoViBe hingegen ähnelt dem Thema dieser Bachelorarbeit am stärksten, da der Fokus auf dem Testen von Visualisierungen liegt. OntoViBe besteht zwar aus mehreren Modulen; ein systematisches Testen einer bestimmten Gruppe von OWL-Elementen ist damit jedoch nicht möglich. Außerdem wird OWL 2 nicht vollständig abgedeckt.

Mit Protégé könnte prinzipiell eine Benchmark-Ontologie erstellt werden, welche außerdem auch auf eine spezielle Ontologievisualisierung angepasst werden kann. Der erforderliche Aufwand ist hierbei jedoch relativ groß und die resultierende Ontologie aufgrund des Entstehungsprozesses stark auf eine einzelne Visualisierung beschränkt.

Mit dem Ergebnis dieser Arbeit soll es möglich sein, eine Ontologievisualisierung und die unterstützten OWL-Konstrukte modular und systematisch zu validieren. Es sollen alle Konstrukte von OWL 2 separat auswählbar sein, um eine größtmögliche Flexibilität beim Erstellen einer Benchmark-Ontologie zu

erhalten. Hierbei sollen aber trotzdem *Module* angeboten werden, die die einzelnen OWL-Konstrukte systematisch zusammenfassen. Im Gegensatz zum manuellen Erstellen mit beispielsweise Protégé, sollen hiermit vordefinierte Testfälle kombinierbar sein, sodass der Gesamtaufwand auf ein Minimum beschränkt ist.

3. Konzept

In diesem Kapitel wird zuerst ein Überblick des Gesamtkonzepts gegeben. In den folgenden Abschnitten wird das Konzept dann in seinen grundlegenden Teilen erklärt und die Entscheidungen, die während dieser Arbeit gefallen sind, begründet.

3.1. Überblick

OntoBench soll dem Benutzer eine Übersicht der Elemente geben, aus denen der Generator eine Ontologie erstellen kann. Die Elemente sollen einzeln auswählbar sein, damit zum einen Visualisierungen getestet werden können, deren Entwicklung sich erst im Anfangsstadium befindet. Zum anderen muss eine Visualisierung auch nicht alle OWL-Konstrukte darstellen können, was zum Beispiel der Fall ist, wenn nur bestimmte OWL-Profile, also Subsets der OWL-Spezifikation, abgebildet werden sollen. Für diese Profile bietet es sich an, dem Nutzer eine Vorauswahl der Elemente zu bieten, die in diesen Profilen enthalten sind. Nach dem Generieren soll man die Ontologie entweder herunterladen können oder eine eindeutige URL erhalten, mit der sie auch über das Internet abgerufen werden kann.

Der Generator soll über eine Weboberfläche ansprechbar sein. In einer Anfrage werden die Benutzereingaben über eine Schnittstelle an den Server geleitet, der daraufhin die Ontologie erstellt und diese in seiner Antwort zurückgibt. Dem Anwender soll die Ontologie anschließend angezeigt werden.

3.2. Ontologie

In den folgenden Abschnitten wird beschrieben, was mit einer generierten Ontologie getestet werden können soll und wie genau sie aufgebaut sein muss.

3.2.1. Umfang

Theoretisch soll mit den Ontologien *alles* getestet werden können, was in Frage kommt. Eine grobe Richtung hierfür ergibt sich aus dem Begriff *Ontologievisualisierungen* aus der Aufgabenstellung. Zum einen sollen Eigenschaften einer Ontologie getestet werden, zum anderen Eigenschaften von möglichen Visualisierungen mit Bezug auf die Ontologie.

Testbarkeit der Ontologie

Ontologien sollen nach Empfehlung des W3C in der Web Ontology Language (OWL) geschrieben werden. OWL bietet vordefinierte Konstrukte, mit denen man Aussagen definieren kann. Durch diese Konstrukte lassen sich unter anderem Klassen definieren (zum Beispiel Menschen), diese mit einer Eigenschaft beschreiben (Menschen können sehen) oder die Eigenschaften genauer einschränken (Menschen haben genau zwei Augen).

In einer Ontologievisualisierung, welche alle verfügbaren Informationen darstellt, müssten nun auch alle Aussagen visualisiert werden. Auf die Beispiele bezogen, müsste man als Nutzer die Gruppe von Menschen erkennen können. Man müsste auch sehen, dass die Menschen eine bestimmte Eigenschaft haben, oder dass eine Eigenschaft eingeschränkt ist. Es ist deshalb grundlegend, dass alle Elemente der Sprache, mit denen ich meine Aussagen formulieren kann, getestet werden können sollen.

Damit wäre aber nur eine Dimension abgedeckt; die Kombination verschiedener OWL-Elemente wäre hierbei nicht berücksichtigt, was aber auch bewusst so sein soll. Würde man spezielle Kombinationen und Strukturen testen wollen, gäbe es zwei Möglichkeiten: Visualisierungen einzelner OWL-Elemente werden miteinander verknüpft oder es entsteht ein neue Darstellung für diese.

Eine besondere Darstellung einer komplexen Struktur kann aber nicht getestet werden, da sie zu spezifisch für eine Visualisierung ist. Hierfür müsste also Details der zu testenden Visualisierungen vorhanden sein, was zur Folge hätte, dass Ontologien von OntoBench nicht mehr allgemein gehalten wären. Deshalb sollen spezielle Kombinationen von OWL-Konstrukten explizit nicht testbar sein.

Testbarkeit der Visualisierung

Bis jetzt wäre die Funktionalität der Visualisierung hinsichtlich der unterstützten OWL-Elemente testbar. Beim Ansatz einer graphbasierten Visualisierung wäre aber vielleicht interessant, wie sich diese mit besonders vielen Knoten oder mit besonders vielen Kanten zwischen zwei Knoten verhält, also wie sie skaliert. Dieser Aspekt soll allerdings nicht im Fokus stehen, da alle verschiedenen Visualisierungskonzepte und die Visualisierungen bekannt sein müssten, um die Skalierbarkeit dieser zu testen.

Der Fokus soll viel eher darauf liegen, bestimmte Fallstricke oder Besonderheiten von OWL mit Bezug zu Visualisierungen testen zu können. Einige Ideen stammen von der Entwicklung an *WebVOWL*, an der ich mitgewirkt habe und diese Fallstricke selbst erleben durfte. Aus diesen Erfahrungen lässt sich aber nicht direkt auf andere Visualisierungen schließen, da manche Testfälle erst durch die graphbasierte Visualisierung von *VOWL* [LNHE14] ermöglicht wurden; sie bilden allerdings ein gutes Fundament.

3.2.2. Feature

Die Begriffe *Element* oder *Konstrukt* beziehen sich bisher hauptsächlich auf OWL. In Ontologien können aber auch kleinere Konstrukte aus mehreren OWL-Elementen verwendet werden, weshalb diese unter einem neuen Begriff zusammengefasst werden sollen. Da OntoBench in der englischen

Sprache verfügbar sein wird, habe ich den kurzen englischen Begriff *Feature* gewählt, der einen Testfall beschreibt. Ein Feature ist ein Bestandteil der generierten Ontologie und sollte seinen Inhalt durch eine sprechende Bezeichnung ausdrücken. *Feature* wird im Folgenden auch als Synonym für ein Element von OWL verwendet, wenn dieses explizit durch das Feature getestet werden können soll.

3.2.3. Anforderungen an die Features

Bei der Auswahl der OWL-Elemente und beim Definieren der Features sollen die folgenden Anforderungen eingehalten werden, um eine möglichst nützliche und aussagekräftige Ontologie zu erhalten.

Kompakte Definition der Features

Ein Feature soll so knapp wie möglich in der Ontologie definiert werden, aber dabei trotzdem seine Anforderungen erfüllen. Dadurch kann die Ontologie beziehungsweise ein einzelnes Feature in der Ontologie einfacher und schneller nachvollzogen werden, als wenn komplexe Konstrukte verwendet werden.

Der Hauptgrund hierfür ist allerdings, dass durch ein zu komplexes Feature eventuell zu viele andere OWL-Elemente, also auch indirekt andere Features, miteinbezogen werden würden. So würde der Anwender zum einen einen unerwartet großen Raum an anderen Features abdecken, der ihm aus der Beschreibung des einen Features vielleicht gar nicht deutlich wurde und den er gar nicht testen will. Zum anderen würden aber vielleicht auch OWL-Elemente miteinbezogen, die von der zu testenden Visualisierung gar nicht unterstützt werden. Dadurch könnte die generierte Ontologie unbrauchbar werden, weil die Visualisierung eventuell die Darstellung komplett verweigert. Die Kombination eines geforderten mit einem unerwarteten Feature könnte aber auch eine andere Darstellung des geforderten Features zur Folge haben.

Unabhängige Definition der Features

Diese Anforderung ist der obigen in der Hinsicht ähnlich, dass unerwartete Nebenwirkungen der Ontologie auf die Visualisierung vermieden werden sollen.

Einzelne Features, die die gleichen OWL-Elemente nutzen, sollen möglichst unabhängig voneinander definiert werden. OWL-Elemente sollen also bewusst mehrfach definiert und einmalig verwendet werden. Dadurch soll es ebenfalls leichter sein, ein Feature in der Ontologie nachvollziehen zu können. Es ergibt sich aber auch der Vorteil, dass unabhängige Features in der Visualisierung voraussichtlich getrennt voneinander dargestellt werden. Dadurch können die eingegebenen Testdaten leichter verifiziert und Fehler in der Visualisierung einfacher dem Feature beziehungsweise den zugrunde liegenden OWL-Elementen zugeordnet werden.

Benennung der OWL-Konstrukte eines Features

Die OWL-Konstrukte, die von einem Feature verwendet werden, sollen keine zufälligen, sondern aussagekräftige IRIs beziehungsweise Namen erhalten. Da diese Namen in der Visualisierung angezeigt werden, sollen sie die Bezeichnung des getesteten Features beinhalten. Beim Validieren der Visualisierung kann somit schnell erkannt werden, ob ein Feature anders als beabsichtigt visualisiert wird und welche Features nicht oder nicht vollständig visualisiert werden.

Werden mehrere OWL-Elemente für ein Feature verwendet, sollen trotzdem alle Elemente die Bezeichnung des Features beinhalten. Wird zum Beispiel eine `owl:ObjectProperty` getestet, sollen auch die beiden umliegenden Klassen zumindest *ObjectProperty* im Namen haben, damit die Zugehörigkeit zum Feature erkennbar bleibt.

In den Ontologien von OntoViBe [HLNE14a] wird das ähnlich gehandhabt. Jedes Konstrukt hat einen Namen, der schnell auf den Testfall schließen lässt. Da OntoViBe allerdings die einzelnen Testfälle stärker miteinander vernetzt, wird in der Visualisierung nicht unbedingt klar, ob alle OWL-Konstrukte auch wirklich zu dem visualisierten Testfall gehören. So wird für den Testfall des `owl:intersectionOf`-Elements unter anderem ein Element mit dem Namen *DeprecatedClass* genutzt, sodass nicht direkt klar ist, ob dieses überhaupt dazu gehört.

Kompakte Darstellung der Ontologie

Je mehr OWL-Elemente in der Ontologie aufgeführt sind, desto mehr Elemente müssen auch visualisiert werden. Durch eine kompakte und knappe Definition eines Features wird auch dessen Visualisierung zumindest in einem gewissem Umfang kompakt gehalten. Um aber die Gesamtmenge der dargestellten OWL-Elemente niedrig zu halten, muss in gewissem Maße die Anforderung nach einer unabhängigen Definition gebrochen werden.

Eine Property ist eine Eigenschaft, die ein OWL-Element mit einem anderen beschreibt. Für jede Property werden mindestens drei Elemente benötigt: Das beschriebene Element, die Property und das beschreibende Element. Wie in OntoViBe [HLNE14a] sollte ursprünglich ein einziges Element gewählt werden, welches alle Properties beschreiben. Hierdurch kann die Gesamtmenge der OWL-Elemente in der Ontologie, die für Properties benötigt werden, um annähernd ein Drittel reduziert werden.

Bei Feldversuchen während und nach der Entwicklung zeigte sich jedoch in manchen Visualisierungen, dass die große Anzahl an Properties, die dasselbe Element beschreiben, die Visualisierung unübersichtlich erscheinen ließ. Im Endeffekt lies sich dann wegen Überdeckungen die Funktionalität der Visualisierung schlechter bewerten. Als Maßnahme dagegen ist die Anzahl, wie oft ein Element beschrieben werden kann, beschränkt. Nach Erreichen der Schranke wird dynamisch ein neues, beschreibendes Element verwendet.

3.2.4. OWL-Features

Wie bereits beschrieben, sollten in jedem Fall alle Elemente der Web Ontology Language zum Generieren einer Ontologie zur Auswahl stehen. Eine ausführliche Auflistung bietet die *OWL 2 Quick Reference*

[BKMPS12]. Die unterstützten OWL-Konstrukte sind in einer Tabelle in Anhang A aufgelistet. Auf dieser Tabelle aufbauend werden im Folgenden nur ausgenommene OWL-Konstrukte erläutert und weitere Testfälle beschrieben.

Ausnahmen

Aufgrund einer fehlenden Implementierung in der verwendeten Programmbibliothek oder fehlender Parametrisierung der Features können manche OWL-Konstrukte nicht verwendet werden.

So wählt die Bibliothek in einem Fall bei zwei verschiedenen, syntaktisch unterschiedlichen Elementen automatisch ein semantisch äquivalentes Element. Mit `owl:differentFrom` kann die Aussage getroffen werden, dass sich ein Individual von einem anderen unterscheidet. Um mehrere Individuals als paarweise verschieden zu deklarieren, wird `owl:AllDifferent` verwendet. Das OWL-Element `owl:differentFrom` kann die Bibliothek mindestens in der Turtle-Syntax nicht ausgeben, dafür allerdings das semantisch äquivalente Gegenstück `owl:AllDifferent`.

Für Konstrukte wie `owl:imports` oder `owl:versionIRI` wären Benutzereingaben dringend notwendig, um hier gültige Werte verwenden zu können. Außerdem kann `owl:onProperties` meines Wissens nach nicht von der verwendeten Programmbibliothek für eine Ontologie berücksichtigt werden.

Mehrsprachige Annotationen

OWL bietet mit Annotationen eine Möglichkeit, wie der gesamten Ontologie oder einzelnen Elementen weitere Informationen angehängt werden können. So kann mit `owl:versionInfo` beispielsweise mitgeteilt werden, in welcher Version die Ontologie vorliegt. Mit `rdfs:label` kann der Ontologie oder einem einzelnen Element eine Bezeichnung gegeben werden.

Wie in der *OWL 2 Syntax* [MPSP12] unabhängig von der Definition der OWL-Elemente beschrieben, soll die Spezifikation durch mehrsprachige Annotationen noch in größerem Umfang abgedeckt werden.

3.2.5. Visualisierungs-Features

Außer für Elemente von OWL sollen noch weitere Features angeboten werden, deren Fokus auf der Kombination von OWL-Konstrukten liegt, die Ontologievisualisierungen Schwierigkeiten bereiten könnten.

Viele Individuals

Eine Klasse kann theoretisch eine beliebig große Anzahl an Individuals haben, die zu dieser Klasse gehören. Bei der Klasse *Mensch* könnte eine Visualisierung beispielsweise mehr als 7 Milliarden Individuals anzeigen, während die Anzahl der Individuals der Klasse *Sinnesorgan* nicht mal den zweistelligen Bereich überschreiten würde. In einer Visualisierung könnten die einzelnen Individuals allerdings auch gar nicht, nur als Zahl oder auch durch visuelle Attribute kenntlich gemacht werden. Daher sollen unterschiedliche Mengen von Individuals testbar sein.

Reflexive Eigenschaften

Eine reflexive Eigenschaft einer Klasse verweist auf die Klasse selbst. So hat beispielsweise jedes Individual der Klasse *Mensch* die Eigenschaft *hatVerwandten*, welche wieder auf die Klasse *Mensch* zeigt.

Bei einer graphbasierten Visualisierung, wird eine Eigenschaft üblicherweise als Kante zwischen zwei Knoten gezeichnet werden, die jeweils Klassen repräsentieren. Wird genauso bei einer reflexiven Eigenschaft vorgegangen, könnte die Kante versteckt sein, da der Start- und Zielpunkt gleich sind. Da sich die Problemstellung bei mehreren reflexiven Eigenschaften einer einzelnen Klasse erschwert, soll es hierfür extra Testfälle geben.

Parallele Eigenschaften

Zwischen zwei Klassen, wie beispielsweise *Angestellter* und *Chef*, können mehrere Eigenschaften bestehen. So wären *arbeitetFür*, *istVerwandtMit* oder *istNachfolgerVon* mögliche Beispiele.

In einer graphbasierten Visualisierung könnten in diesem Fall mehrere Kanten zwischen zwei Knoten verlaufen, wodurch es unter anderem zu Überdeckungen kommen könnte. Damit diese Fälle testbar sind, soll es möglich sein, eine unterschiedliche Anzahl „paralleler“ Eigenschaften zu testen.

Weitere Annotationen

In der OWL-Spezifikation sind, wie in einem vorigen Abschnitt beschrieben, bereits einige Annotationen vordefiniert. Es können aber ebenfalls eigene oder an anderer Stelle definierte Annotationen verwendet werden.

Ein prominentes Beispiel sind die *DCMI Metadata Terms* [DCM12]. Sie bieten Annotationen wie `dc:creator`, um den Erzeuger einer Ressource zu beschreiben, oder `dc:description`, mit der eine Ressource genauer beschrieben werden kann. So könnte damit eine mit `owl:versionInfo` annotierte Ontologie um Versionshinweise ergänzt werden und Änderungen zu den vorigen Versionen aufzeigen.

Wird eine Ontologie nun visualisiert, können auch die Annotationen besonders dargestellt werden. So ist zum Beispiel wichtig zu wissen, wenn eine Ontologie oder Konstrukte von OWL-Elementen durch

die Annotation `owl:deprecated` als veraltet markiert sind. Mit weiteren, nicht in OWL 2 definierten Annotationen kann die Visualisierung weiter ergänzt werden.

3.2.6. OWL-Profile

Wie bereits in den Grundlagen in Unterabschnitt 2.1.4 vorgestellt, definieren OWL 1 und OWL 2 sogenannte Profile, die Untermengen der in der Spezifikation beschrieben Sprache sind.

Eine Ontologie, die sich nach den Profilen OWL Lite oder OWL DL richten soll, kann relativ einfach erstellt werden. OWL DL baut auf OWL Lite auf und stellt dadurch eine einfache Erweiterung dar. Im Gegensatz zu den Einschränkungen von OWL Full, enthalten sie Regeln bezüglich der erlaubten OWL-Elemente und eine Einschränkung eines einzelnen Wertebereichs. Da sich mit OWL DL aber bereits alle Elemente der OWL 1 Spezifikation abbilden lassen und OWL Full gewisser weise nur Restriktionen bezüglich dem Aufbau von Konstrukten enthält, soll OWL Full nicht als Profil angeboten werden. Diese Entscheidung ist konsistent mit der Definition des Umfangs, nach der keine Strukturen mit der generierten Ontologie getestet werden können müssen.

OWL EL, QL und RL sind im Vergleich zu den vorigen Profilen wesentlich komplexer, da sie nicht nur binär angeben, welche Elemente im Profil enthalten sind. So existieren hier Einschränkungen, die die Kombination von zwei oder mehr OWL-Elementen nur zulassen, wenn diese bestimmte Werte haben. Im Gegensatz zu den Profilen von OWL 1 bauen die Profile von OWL 2 auch nicht aufeinander auf. Dadurch treten teilweise widersprüchliche Einschränkungen bezüglich desselben Elements der Spezifikation auf. Folglich können mit in Konflikt stehenden Features nicht alle Profile abgedeckt werden, weshalb separate Features für diese benötigt werden.

Um Duplikate der Features zu vermeiden, werden nicht für jedes Profil alle Features separat angelegt. Features für ein OWL-Element werden nur dann doppelt angelegt, wenn dort ein Konflikt besteht. Falls ein Feature in einem Profil ohne Einschränkungen und im anderen nur mit Einschränkungen verwendet werden darf, wird dieses an die Einschränkung angepasst und anschließend von beiden Profilen rechtmäßig verwendet.

3.2.7. Syntax

Damit die generierte Ontologie von einer Visualisierung oder von anderen Anwendungen genutzt werden kann, muss sie in ein Datenformat gebracht werden, das diese interpretieren können. Das Datenformat ist hierbei durch die Syntax einer Ontologie gegeben. Im *OWL 2 Overview* [W3C12] sind verschiedene Syntaxen genannt, in welche eine Ontologie serialisiert werden kann.

Nach diesem OWL-Dokument muss *RDF/XML* von allen Tools unterstützt werden, die mit OWL 2 arbeiten. Es dient dabei als das primäre Austauschformat für Ontologien. Deshalb ist es essentiell, dass die Anwendung eine Ontologie in dieser Syntax ausgeben kann.

Während der Implementierung zeigte sich, dass Ontologien mit sehr geringem Aufwand in einer der Syntaxen aus Tabelle 3.1 ausgegeben werden können. Aus diesem Grund stehen in der Anwendung alle genannten zur Auswahl.

Syntax	Unterschiede im Vergleich zu RDF/XML
OWL/XML	Einfacher von Anwendungen zu verarbeiten, die mit XML arbeiten können.
Functional Syntax	Bringt die formale Struktur besser zum Ausdruck.
Manchester Syntax	Einfacher, um Ontologien lesen oder schreiben zu können.
Turtle	Einfacher, um RDF-Tripel lesen oder schreiben zu können.

Tabelle 3.1.: Vergleich der obligatorischen RDF/XML-Syntax mit anderen, optionalen Syntaxen für Ontologien.

Standard-Syntax

Wenn eine Ontologie über eine URL abgerufen wird, muss standardmäßig auf jeden Fall *RDF/XML* angeboten werden; andere Syntaxen können optional auch zur Verfügung stehen. Wenn die Ontologie generiert und anschließend angezeigt wird, würde sich nach dem *OWL 2 Overview* die *Manchester Syntax* beziehungsweise *Turtle* hierfür eher eignen, siehe Tabelle 3.1.

Mit der *Manchester Syntax* waren allerdings im Gegensatz zu *Turtle* alle IRIs ausgeschrieben, worunter die Lesbarkeit meiner Meinung nach stark litt. Da im Vergleich dazu die ausgegebene Ontologie in *Turtle* für den langen, gemeinsamen Teil der IRI einen Präfix verwendet und die Ontologie deshalb lesbar blieb, sollte *Turtle* die Standardsyntax für die Anzeige in der Oberfläche sein.

Syntax eines OWL-Element in der Übersicht

Für Features von OWL-Konstrukten soll die Bezeichnung dem testbaren Konstrukt entsprechen. Ebenso wie die ganze Ontologie kann ein einzelnes OWL-Element in verschiedenen Syntaxen unterschiedlich aussehen.

Die *Functional Syntax* würde sich zur Anzeige der OWL-Elemente eignen, da sie in den meisten OWL 2-Dokumenten verwendet wird und somit bekannt ist. Für die Anzeige der einzelnen Elemente ist sie allerdings ungeeignet, da Elemente aus anderen Syntaxen nicht unbedingt auf sie abbildbar sind. So wird mit `AllDisjointClasses` ausgesagt, dass eine beliebige Menge von Klassen verschieden sind. Im Gegensatz hierzu unterscheidet *RDF* zwischen `owl:disjointWith` für zwei Klassen und `owl:allDisjointClasses` für eine beliebige Anzahl.

Damit alle OWL-Konstrukte und nicht nur die in einer Syntax vorhandenen testbar sind, soll bei der Anzeige der OWL-Elemente auf *RDF* gesetzt werden.

3.2.8. URL und IRI

Da die generierten Ontologien über das Internet abgerufen werden können sollen, müssen sie eine URL besitzen, die auf sie zeigt. Beim Generieren kann dem Server in extra Datenpaketen noch mitgeteilt werden, welchen Inhalt die Ontologie haben soll. Im Gegensatz dazu ist beim Abruf über eine URL diese die einzige Informationsquelle für den Server. In der URL müssen also alle Informationen codiert sein, die der Server braucht, um die Ontologie zu bestimmen. Mit derselben URL soll auch immer die

gleiche Ontologie abgerufen werden können. Die URL der Ontologie kann außerdem auch als IRI der Ontologie verwendet werden.

Die URL wird immer mit einem konstanten Teil beginnen, welcher auf den Webserver verweist, von dem die Ontologien abgerufen werden können. Ein Beispiel wäre die folgende Adresse: `http://www.ontologie-generator.de/generated-ontologies/`. Dieser Teil hängt zum einen von der verfügbaren Internetadresse und den Laufzeitbedingungen des Servers ab. Zum anderen enthält er keine genauen Informationen, mit denen auf eine spezielle Ontologie geschlossen werden könnte. Daher soll hier nur der variable Teil der URL erklärt werden, durch den eine Ontologie identifiziert werden kann.

Features als Parameter

Um leicht von einer URL auf die generierte Ontologie zu schließen, ist es am einfachsten, alle enthaltenen Features der Ontologie darin zu codieren. So könnte man jedem Feature eine eindeutige Nummer zuweisen und dann die Nummern der zu generierenden Features in der URL konkatenieren. Da so eine Adresse aber nicht wirklich sprechend wäre, sollen kurze Namen der Features anstelle von Zahlen verwendet werden: `http://.../?features=owlclass,functionalproperty,loops,owlthing`.

Bei gut gewählten Namen kann bereits mit der URL der Inhalt der Ontologie grob bestimmt werden. Außerdem kann auch leicht die Liste der Features reduziert werden, falls einzelne Features doch nicht benötigt werden und die Länge der URL überschaubar ist. Sobald der Server dann eine Anfrage über diese Adresse erhält, kann er die Ontologie mit den gewählten Features erstellen.

Der Server müsste bei dieser Methode weder Informationen zu generierten Ontologien noch die Ontologien selbst speichern. Als Nachteil könnte sich herausstellen, wenn die Generierung aus der URL durch eine große Anzahl an Aufrufen eine zu starke Rechenlast erzeugt, die den Server im ungünstigsten Fall überlastet. Gegebenenfalls wäre dann aber eine Art Zwischenspeicher eine Lösung.

Kürzere URL

Von meinen Betreuern wurde bei einem Treffen angesprochen, dass eine URL, welche alle Features enthält, schlecht in sozialen Netzwerken oder auf Plattformen wie Twitter geteilt werden kann. Dieses Argument ist allerdings nicht allgemeingültig für alle Plattformen. Twitter verwendet für alle URLs einen eigenen Linkkürzer, der unabhängig von der ursprünglichen URL immer Adressen mit einer Länge von 22 Zeichen[1] erzeugt. Facebook zeigt lange URLs verkürzt an und besitzt ebenfalls einen eigenen Linkverkürzer, der auf Mobilgeräten Einsatz findet.[2] Da bei E-Mails und anderen Kommunikationsmitteln wie beispielsweise Chats eine solche Linkverkürzungsfunktion meist aber nicht vorhanden ist, soll eine Ontologie auch mit einer kürzeren URL angeboten werden.

[1] https://support.twitter.com/articles/475280
[2] https://www.facebook.com/notes/the-next-web/fbme-facebooks-new-url-shortener/204973923321

Bei etwa 50 ausgewählten Features und einer durchschnittlichen Länge eines Bezeichners von zehn Zeichen wäre allein der dynamische Teil der URL bereits 500 Zeichen lang. Dazu müssten noch die Anzahl der Trennzeichen und die Länge des vorderen Teils addiert werden.

Mit kleingeschriebenen, alphanumerischen Bezeichnern und einer festen Länge von zwei Zeichen, könnten immerhin $(26 + 10)^2 = 1296$ Features identifiziert werden. Um weiter zu komprimieren, könnten durch die bekannte Länge der Bezeichner die Trennzeichen ausgelassen werden. Doch selbst hier hätte der dynamische Teil der URL eine Länge von 100 Zeichen.

Folglich soll die URL noch stärker verkürzt werden, wodurch sie weniger Informationen enthalten kann. Wenn die URL den Inhalt einer Ontologie nicht vollständig definieren kann, müssen die verlorenen Informationen an einer anderen Stelle gespeichert werden – also auf dem Server.

Nach einer ersten Idee soll nach jedem Generieren eine aufsteigende numerische ID mit den verwendeten Features in einer Datenbank gespeichert werden. Die ID kann anschließend in der URL verwendet und vom Server zu den Features decodiert werden. Der Vorteil liegt klar auf der Hand: So können beispielsweise mit einer 6-stelligen ID eine Millionen Ontologien identifiziert werden. Die URL wäre damit drastisch verkürzt, was an diesem Beispiel deutlich gemacht wird: `http://.../id/265/`.

Nach einer ersten Implementierung ist die Vorhersagbarkeit anderer IDs aufgefallen, da es sich bei der ID nur um eine inkrementierte Zahl handelt; das wäre bei dieser Anwendung aber nicht weiter schlimm. Ein größeres Problem ist, dass zwei Ontologien mit den gleichen Features zwei unterschiedliche IDs haben können. Da voraussichtlich OWL-Profile, Ontologien mit allen und mit keinen Features häufig getestet werden, würde die Größe der IDs nur unnötig aufgebläht werden, wenn für diese jedes Mal eine neue ID angelegt werden würde. Daher sollen Duplikate vermieden werden und identische Ontologien die gleiche ID erhalten.

Lange IRIs und Probleme in Tools

Standardmäßig wurde immer die lange URL zu einer Ontologie mit einer langen IRI angeboten; die IRI glich also der URL. Die kurze URL sollte explizit angefordert werden, wenn sie benötigt wurde, um Speicher auf dem Server zu sparen. Mit der langen URL und IRI traten aber Bugs wie über den Bildschirmrand ragende Textfelder in Tools wie Protégé auf, welche die IRI der Ontologie anzeigten. Daher werden lange und kurze URLs und IRIs wie folgt verwendet:

- Die lange URL bleibt verfügbar, damit erkennbar ist, was in der Ontologie enthalten ist.
- Die kurze URL bleibt verfügbar, damit die Ontologie besser geteilt werden kann.
- Die Ontologie verwendet als IRI immer die kurze URL, um Tools und Visualisierungen bei der Anzeige zu entlasten.

Angabe der Syntax

Bis jetzt ist die Auswahl der Syntax der generierten Ontologie völlig übergangen worden. Ohne eine explizite Angabe soll *RDF/XML* verwendet werden, da diese Syntax von allen OWL 2 Tools unterstützt werden muss.

Die Syntax der Ontologie soll durch die zugehörige Dateiendung in der URL angegeben werden. Dadurch kann der Anwender auf einen Blick sehen, in welchem Format ihm die Ontologie vorliegt und kann sie gegebenenfalls schnell ändern. Wenn die Ontologie lokal gespeichert wird, kann sie durch die richtige Dateiendung auch direkt mit etwaigen Standardprogrammen geöffnet werden.

Die URL soll wie folgt aussehen: `http://.../id/265.owl`. In diesem Fall signalisiert `.owl`, dass die *RDF/XML*-Syntax benötigt wird.

3.3. Architektur

Die Entscheidung für eine Webanwendung fiel relativ schnell. Zum einen sollte - wie bereits oben erwähnt - die generierte Ontologie auf jeden Fall über eine URL abgerufen werden können. Dafür müsste die Anwendung eine Netzwerkschnittstelle beispielsweise zum Internet bieten.

Für eine Weboberfläche spricht auch, dass diese von praktisch jedem möglichen Nutzer aufgerufen werden kann. Aufgrund des Themengebiets werden die allermeisten Nutzer einen Computer oder ein ähnliches Gerät mit einem Internetanschluss besitzen. Falls der Zugriff zum Internet nicht dauerhaft verfügbar ist, kann man eine generierte Ontologie immer noch herunterladen und lokal testen. Alternativ könnte man die Anwendung auch bei sich lokal starten, um auch offline die Möglichkeit zu haben, eine Ontologie zu generieren.

Aus technischer Sicht bietet der Zugriff über eine Internetseite dem Nutzer den Vorteil, dass er nicht die komplette Anwendung herunterladen und installieren muss. Gegebenenfalls müssten davor sogar noch Abhängigkeiten oder Laufzeitumgebungen wie beispielsweise Java oder das .NET-Framework installiert werden, damit sie überhaupt gestartet werden könnte. Mit Eingabe der Internetadresse oder einem Klick auf einen Link, der auf die Webseite verweist, ist der Anwender bereits in der Lage, OntoBench in vollem Umfang zu nutzen.

Da der Webbrowser die Laufzeitumgebung stellt, muss man als Entwickler hierbei auf die große Anzahl verschiedener Umgebungsbedingungen Rücksicht nehmen. So können zum Beispiel Mozilla Firefox, Google Chrome oder Microsoft Edge in unterschiedlichsten Versionen verwendet werden, welche die Webseiten meist nicht exakt gleich darstellen. Weiterhin können diese auf unterschiedlichen Betriebssystemen wie Microsoft Windows, Mac OS X oder einer der vielen Linux-Distributionen wiederum auch in unterschiedlichen Versionen laufen. Diese können sich zum Beispiel in den verfügbaren Schriftsätzen unterscheiden, weshalb die Texte auf der Webseite eventuell unterschiedlich aussehen würden. Anhand dieser kleinen Liste von Faktoren sieht man schnell, dass bei der Entwicklung einer Webanwendung auf jeden Fall Wert auf die Kompatibilität gelegt werden muss.

Auf Entwicklerseite bringt eine Weboberfläche allerdings auch einige Vorteile mit sich. Die Gestaltung eines ansehnlichen Internetauftritts ist mit Verwendung eines gut ausgestatteten Frameworks relativ

einfach. Diese Frameworks bieten dem Entwickler meist eine große Anzahl durchdachter und auch vom Stil her zueinander passender Komponenten, aus denen die Oberfläche aufgebaut werden kann.

Als wohl bekanntestes Beispiel eines solchen Frameworks gilt *Bootstrap*[3] von Twitter. Deren Entwickler haben zum Beispiel darauf geachtet, dass verschiedene Browser dessen Elemente möglichst gleich darstellen. Das Framework bietet außerdem auch ein System an, mit dem sich das Design der Webseite automatisch an verschiedene Displaygrößen anpassen lässt und somit nicht nur an normalen Desktop-PCs, sondern beispielsweise auch an Smartphones bedient werden kann. Dadurch kann man als Entwickler einen größeren Fokus auf ein stimmiges Design und die Implementierung der Logik der Webanwendung legen.

Ein weiterer Aspekt für mich ist, dass ich bereits Erfahrung bei der Entwicklung von Webanwendungen habe und mit den Technologien vertraut bin. Da ich an der Universität, in meiner Freizeit und bei der Arbeit mit Webtechnologien beziehungsweise einer Webanwendung arbeite, kann ich den Implementierungsaufwand hierfür relativ gut abschätzen.

3.4. Weboberfläche

Beim Aufruf der Weboberfläche soll der Nutzer nicht überfordert werden und eine Vorstellung bekommen, wie er diese bedienen kann. Um die Oberfläche einfach zu gestalten, sollen die Auswahl der Features und das Generieren der Ontologie getrennt angezeigt werden. Es soll außerdem eine Einführung geben, die dem Anwender einen Überblick über die Funktionen bietet.

Die Aufspaltung der Weboberfläche in die zwei genannten Bereiche ist möglich, da sie unabhängig voneinander genutzt werden können: Solange man auswählt, was in der Ontologie enthalten sein soll, legt man ausschließlich deren Inhalt fest. Im Anschluss daran wird die Form der Ontologie festgelegt, diese dann generiert und dem Nutzer die URL und die Ontologie angezeigt.

Diese Idee hätte man umsetzen können, indem man nach dem Bestätigen der Auswahl auf eine andere Seite zum Generieren weiterleitet, ähnlich wie es üblicherweise bei Anmeldeformularen auf Internetseiten funktioniert. Der Nachteil hierbei ist allerdings, dass man nicht wieder „kurz“ zur Auswahl zurückspringen kann, um sie zu ändern, weil ein neuer Seitenaufruf stattfindet. Außerdem müssten die gewählten Features bei jedem Seitenwechsel mitgesendet werden, damit die Anwendung weiß, was generiert werden soll beziehungsweise was ausgewählt ist. Daher soll die Webanwendung nur aus einer einzigen Seite bestehen, die einen offensichtlichen Weg zum Wechsel zwischen den verschiedenen Inhalten bietet.

3.4.1. Feature-Übersicht

Der Anwender soll eine Übersicht über die Features, also die Elemente, aus denen der Generator eine Ontologie erstellen kann, erhalten. Wie beispielsweise in der OWL-Spezifikation getan, kann jedes OWL-Element einer Kategorie zugeordnet werden. Aufgrund der relativ großen Anzahl an Features

[3] http://getbootstrap.com/

macht es Sinn, diese in den jeweiligen Kategorien zu gruppieren. Durch Verwendung einer bekannten Anordnung kann es dem Benutzer somit später leichter fallen, sich in der Übersicht zurechtzufinden. Diese Kategorien wurden für OntoBench leicht angepasst, um ähnliche OWL-Konstrukte besser zu gruppieren.

Ein Ansatz, die Features in Kategorien anzuzeigen, war die Verwendung einer Baumstruktur. Hätte man diese verwendet, könnte die Weboberfläche eventuell sogar auf eine einzige Seite gebracht werden. Man kann sich das wie beim Dateiexplorer von Windows vorstellen, nur dass statt der Ordnerstruktur links die Elemente in einem Baum dargestellt wären und im restlichen Teil die Oberfläche zum Generieren sein würde. Die Vorteile sind klar: Eine kompaktere Oberfläche, mit der außerdem auch schneller die Auswahl geändert und neu generiert werden kann.

Für mich überwiegen die Nachteile allerdings deutlich. Das beginnt bereits beim Definieren des Anfangszustandes der Baumdarstellung. Möglich wäre, dass entweder alle Kategorien geschlossen oder geöffnet sind. Wären sie geöffnet, wäre die Liste wegen der großen Anzahl an Elementen sehr lang. Wären sie geschlossen, müsste man erst alle Teilbäume aufklappen, bevor man ein Feature anwählen könnte. Da in diesem Fall die einzelnen Features nicht sichtbar sind, kann der Anwender auch nicht wissen, wonach genau er überhaupt suchen kann.

Unabhängig von der Darstellung muss der Anwender auch eine Auswahl tätigen können. Da das entweder an den Elementen selbst oder auch durch das Hinzufügen einer Vorauswahl geschehen könnte, wäre die Ansicht schon ziemlich überladen. Dazu kommt auch noch, dass das Layout auf kleineren Bildschirmen kaum skaliert. Der Platz würde vielleicht reichen, um die Bedienelemente für das Generieren neben den Baum zu setzen; um die generierte Ontologie zusätzlich anzuzeigen, wäre es meiner Meinung nach zu eng. Wenn man die zwei Bereiche vertikal nebeneinander anordnet, wäre das Platzproblem gelöst; der Vorteil der kompakten Darstellung hätte sich aber damit erledigt. Daher habe ich mich für ein simpleres Layout entschieden, welches auch auf Mobilgeräten, auf welchen man eventuell auch seine Visualisierung testen möchte, bedienbar bleibt.

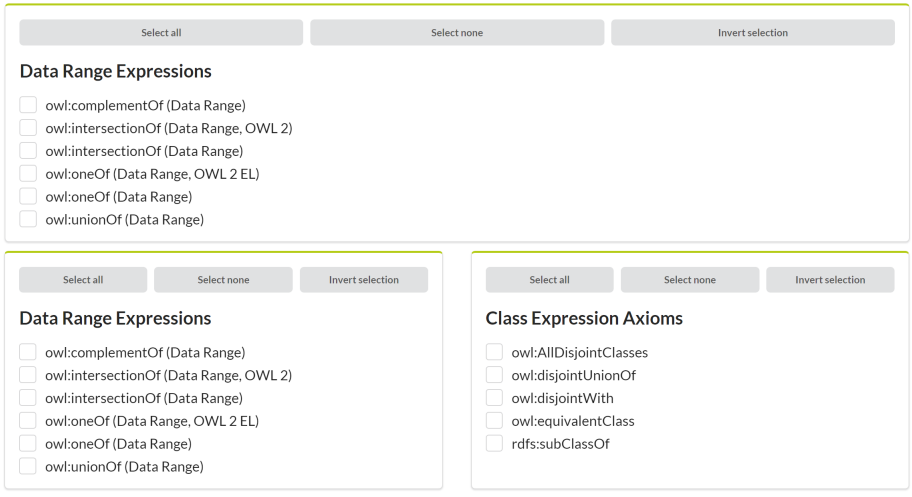

Abbildung 3.1.: Im oberen Teil der Abbildung ist deutlich erkennbar, dass Raum ungenutzt bleibt. Durch eine zweispaltige Ansicht wird dieser besser genutzt.

Als einfache Alternative sollen die Kategorien mit ihren Features auf einer eigenen Seite angezeigt werden. Nach einer ersten Idee sollten diese dann untereinander aufgelistet werden. Während der Implementierung zeigte sich aber, dass die Webseite in der Breite nicht wirklich gefüllt war und ein großer Teil ungenutzt blieb. Daher werden die Kategorien in einer zweispaltigen Tabelle dargestellt. Die einzelnen Zellen bestehen aus dem Titel der Kategorie und der Auflistung der Features. Im Gegensatz zu den Kategorien ist diese Liste alphabetisch sortiert. Durch diese Sortierung lassen sich bestimmte Elemente schneller finden als in einer unsortierten Liste, bei der im schlechtesten Fall alle Elemente angeschaut werden müssen, um das zu finden, was man sucht (Hicksches Gesetz).

3.4.2. Presets

Wie schon mehrfach erwähnt, soll es auch möglich sein eine Vorauswahl an Elementen hinzuzufügen. Hier bieten sich ohne Frage die Profile von OWL an. Sie sollen in der Oberfläche prominent dargestellt werden, da es schneller ist, eine grobe Auswahl zu treffen und diese dann mit wenig Aufwand an seine Vorstellung anzupassen, als alle Elemente einzeln auszuwählen. Das gilt natürlich nur, wenn eine Vorauswahl vorhanden ist, die zumindest grob die gewünschten Elemente enthält. Die Auswahl soll daher auch vor der Tabelle der Features platziert werden und etwas hervorgehoben sein.

Die Funktionsweise für das Anwählen der Elemente nach Auswahl eines Presets war längere Zeit unklar, da die folgenden Möglichkeiten zur Auswahl standen.

Auswahl über Schalter

Meine erste Idee war eine Art Schalter, den der Nutzer an- und abwählen kann, um dadurch die zugehörigen Features an- beziehungsweise abzuwählen.

Was sollte aber passieren, wenn ein paar Elemente bereits vom Nutzer angeklickt wurden: Sollten manuell veränderte Elemente nicht mehr durch die Vorauswahl verändert werden? Falls man sich hierfür entscheidet, müsste man transparent machen, dass die Vorauswahl nicht vollständig angewendet werden konnte. Wenn aber mehrere Schalter aktiv gesetzt werden würden, verliert man als Anwender schnell die Übersicht, welche Vorauswahl denn inwiefern ausgewählt ist. Der Einfachheit halber könnte man aber immer die vorherigen Benutzereingaben überschreiben, da die Auswahl eines Presets auch eine Benutzereingabe ist.

Was passiert, wenn der Nutzer manuell exakt die Elemente auswählt, die eine Vorauswahl bilden: Sollte der Schalter dann automatisch gesetzt werden? Die automatische Auswahl könnte sehr nützlich sein, da der Anwender sehen kann, wie er seine manuelle Auswahl vereinfachen kann. Falls die Auswahl im Anschluss noch erweitert werden würde, müsste der Schalter des Presets aber wieder abgewählt werden, da sonst impliziert würde, dass das Preset die Auswahl immer noch abbildet. Für den Fall, dass die Auswahl die Vereinigung von Vorauswahlen bildet, müsste auch noch ein Verhalten definiert werden.

Die Fragestellung, an dem die Wahl von Schaltern scheiterte, ist aber die folgende: Was passiert, wenn zwei Presets sich überschneiden und ein Konflikt entsteht, weil ein Preset ein Feature verbietet, ein anderes dieses aber benötigt? Eine Möglichkeit wäre, dass Schalter, die Konflikte erzeugen, nicht

gleichzeitig ausgewählt werden dürfen. Hierfür müsste zuerst für jedes Paar von Vorauswahlen bekannt sein, ob diese miteinander verwendet werden können. Dann müsste wie beim ersten Problem dem Anwender in der Oberfläche erklärt werden, was möglich ist und was Probleme verursacht. Meiner Meinung nach verkompliziert das aber den Prozess unnötig, weshalb ich diese Lösung nicht in Betracht ziehen wollte.

Auswahl über einfache Schaltfläche

Nach Überdenken der gesamten Funktion kam ich auf die folgende, relativ simple Lösung: Es soll nur möglich sein, über eine Schaltfläche Elemente zur Auswahl hinzuzufügen.

Eine Vorauswahl soll nur dazu dienen, dem Anwender die Auswahl der einzelnen Elemente zu vereinfachen, indem sie vorschlägt, welche Gruppe davon er generieren und testen könnte. Fügt man mehrere Vorauswahlen hinzu, möchte man meiner Meinung nach auch alle davon gleichzeitig testen. Dabei sollte dem Nutzer aber klar sein, dass die bisherige Auswahl durch eine Vorauswahl nur erweitert wird. Hat der Nutzer bereits manuell Features angewählt, wird durch Hinzufügen der Vorauswahl die Menge der Features also nur erweitert. Hat der Nutzer davor Features abgewählt, dann sollen diese wieder angewählt werden. Die Auswahl eines Presets ist wie bereits oben beschrieben eine Benutzereingabe, die eben auch vorherige Benutzereingaben überschreiben kann. Solange der Name einer Vorauswahl auch gut beschreibt, was dazu gehört, wird das erneute Anwählen eines einzelnen Features für den Nutzer auch nicht überraschend sein.

Grundlegend ist aber in jedem Fall, dass man als Benutzer erkennen können soll, dass es sich um eine Vorauswahl handelt und diese eine Auswahl hinzufügt.

3.4.3. Generator

Die Oberfläche zum Generieren der Ontologie ist der dritte und letzte Teil der Weboberfläche. Zu dieser gehört auch, dass der Anwender eine generierte Ontologie zum einen herunterladen und zum anderen über einen Hyperlink über das Internet aufrufen kann.

Einstellungen und Generieren

Bevor die Ontologie generiert wird, sollte die Syntax ausgewählt werden können, in der sie angeboten werden soll. Als Syntax soll wie bereits beschrieben standardmäßig Turtle angeboten werden; die anderen Syntaxen sind aber ebenfalls auswählbar.

Da der Anwender an dieser Stelle angegeben hat, was in der Ontologie enthalten sein soll und in welcher Form sie ausgegeben wird, kann er die Generierung nun starten. Initial sollte man nun auf eine Schaltfläche klicken, um den Prozess anzustoßen.

Während der Entwicklung hat sich aber gezeigt, dass der lokale Server sehr schnell auf die Anfrage reagierte, die Ontologie generierte und diese in der jeweiligen Syntax zurück gab. Nach ein paar Messungen mit den Entwickler-Tools in Mozilla Firefox und Google Chrome dauerte das Generieren

bei verschiedenen Auswahlen und beim Wechseln der Syntax in jedem Fall weniger als 60ms. Nach maximal 100ms war die generierte Ontologie dann auch in der Oberfläche dargestellt.

Da die Ergebnisse von einer lokalen Anwendung auf einer relativ aktuellen Entwicklermaschine aber nicht wirklich repräsentativ für den Produktiveinsatz der Anwendung sind, bei dem die Anfragen über das Internet gesendet werden und der Server stärker ausgelastet sein kann, wiederholte ich den Versuch mit einem passenderen Aufbau. Die komplette Anwendung lief dann bei *Heroku*[4] auf einem Server mit dem kleinsten Ressourcenpaket. Über eine vergleichsweise schlechte WiFi-Verbindung ist die Ontologie trotzdem noch nach durchschnittlich 500ms angezeigt worden.

Daher wird die Ontologie jedes mal automatisch generiert, sobald der Anwender zum Generator gewechselt oder eine Einstellung in dieser Oberfläche vornimmt.

Abrufen der URL der Ontologie

Wie zu Beginn dieses Abschnitts bereits erwähnt, soll nach dem Generieren die Ontologie und ein Link zu dieser angeboten werden. Der Link soll noch im oberen Bereich der Webseite über der Ontologie stehen, da diese gegebenenfalls sehr lang werden kann. Um den Link schnell an einer anderen Stelle verwenden zu können, hätte sich eine Funktion angeboten, mit der man den Link auf Knopfdruck in die Zwischenablage kopieren könnte. Da diese Funktion allerdings nur mit Hilfe von einem extra *Adobe Flash*-Plugin umgesetzt werden könnte und sich diese Technologie auf dem absteigenden Ast befindet, habe ich die Funktion nicht eingebaut.

Erst im Juli 2015 wurden nach mehreren größeren Sicherheitslücken für kurze Zeit alle in *Adobe Flash* geschriebenen Plugins in Mozilla Firefox blockiert, da diese Schadcode auf die Rechner der Benutzer hätten schleusen können.[5] Ab September 2015 wird *Flash* zumindest in Google Chrome teilweise pausiert werden [Ber15]. Damit wäre die Funktion nicht ohne Weiteres einsetzbar. Eventuell könnten die Warnmeldungen der Browser sogar die Anwender irritieren und schlimmstenfalls abschrecken.

Deshalb wurde die folgende Alternativlösung gewählt: Der Anwender muss nur das Eingabefeld auswählen und kann den markierten Link entweder über das Kontextmenu oder auch mit dem Standard-Tastenkürzel Strg + C kopieren.

Dem Anwender sollen außerdem eine lange und kurze URL zur Ontolgie angezeigt werden, vergleiche Abschnitt 3.2.8. Gegen das gleichzeitige Anzeigen der ursprünglichen, längeren und der kürzeren Internetadresse spricht für mich, dass dadurch nur unnötig Platz in der Oberfläche verbraucht würde, da eher selten beide Adressen benötigt werden. Unvertraute Anwender könnten auch etwas verwirrt werden, da vielleicht nicht direkt klar ist, ob die zwei unterschiedlichen URLs auf dieselbe Ressource, also dieselbe Ontologie, zeigen. Daher soll in der Oberfläche in der Nähe der angezeigten URL ein Schalter platziert sein, mit der sich zwischen der langen und der kurzen URL wechseln lässt.

[4] Heroku bietet eine Plattform für Cloud-Anwendungen, mehr dazu in Abschnitt 4.1.3.
[5] https://addons.mozilla.org/de/firefox/blocked/p946

Herunterladen der Ontologie

Wie bei der URL bietet es sich an, die Schaltfläche zum Starten des Downloads über der angezeigten Ontologie zu platzieren. Theoretisch hätte man als Anwender die Ontologie auch über die bereitgestellte URL anzeigen und beispielsweise im Browser herunterladen und abspeichern können. Durch die extra Schaltfläche wird dem Anwender aber an dieser Stelle die Bedienung erleichtert und die Möglichkeit zum Herunterladen noch einmal explizit gezeigt.

Anzeigen der Ontologie

Die letzte Komponente der Oberfläche des Generators soll die Anzeige der generierten Ontologie sein. Der Anwender soll die Ontologie sehen können, ohne sie zuerst herunterladen zu müssen. So kann er zum einen direkt überprüfen, was genau generiert wurde und sich ein Bild davon machen. Zum anderen kann er so aber auch schnell sehen, wonach genau er in der Visualisierung schauen müsste (beispielsweise nach dem genauen Namen eines Elements). Dieser Fall könnte vor allem dann auftreten, wenn er die URL der Ontologie in die zu testende Visualisierung eingespielt hat und die Ontologie selbst somit nicht mehr vorliegt. Wie auch die URL soll die Ontologie nach einer einfachen Auswahl markiert werden, damit sie direkt kopiert werden kann.

Für die Anzeige hätte sich noch angeboten, den Inhalt durch Hervorheben der vordefinierten Syntax lesbarer zu machen. So könnten beispielsweise Schlüsselworte beim Scrollen über die Ontologie schneller ins Auge springen und der Aufbau der Ontologie vielleicht auch schneller ersichtlich sein. Aufgrund fehlender Bibliotheken, die die spezielle Syntax einer Ontologie unterstützen würden, konnte diese Idee leider nicht umgesetzt werden. Um eine strukturiertere Ansicht der Ontologie zu erhalten, bietet sich aber sowieso eher *Protégé* oder ein anderer Ontologie-Editor an. Durch die fachliche Unterstützung in der Anwendung kann der Benutzer die Ontologie nach dem Download auch besser als in einem einfachen Texteditor anpassen.

3.5. Server

Im Gegensatz zur Weboberfläche fällt das Konzept für den Teil der Anwendung, die auf dem Server läuft, bewusst wesentlich schlanker aus.

Die Serverseite läuft bei Betrieb der Anwendung im Hintergrund. Solange die Weboberfläche und die dort angebotenen Daten, welche in den vorigen Abschnitten beschrieben wurden, spezifiziert sind, ist der Server prinzipiell austauschbar. Der Server wird gewissermaßen indirekt spezifiziert: Zum einen von außen über die Schnittstelle zur Weboberfläche, zum anderen von innen über die erwartete Funktionalität und die Daten, die er generieren muss.

3.5.1. Anforderungen an den Server

Die Anforderungen an den Server können den folgenden, abstrakten Komponenten zugeordnet werden.

Webschnittstelle

Der Server muss auf jeden Fall eine Webschnittstelle anbieten. Über die soll die Weboberfläche mit den dynamischen Daten, wie beispielsweise den verfügbaren Features, die der Generator im Server unterstützt, befüllt werden. Außerdem werden über diese Schnittstelle die einzelnen Ontologien zum Beispiel aus den getesteten Visualisierungen heraus abrufbar sein.

Die Weboberfläche soll völlig dynamisch aufgebaut sein, damit unter anderem die unterstützen Features oder die Vorauswahlen nicht an zwei Stellen gepflegt werden müssen. Wenn der Server und die Oberfläche sehr stark entkoppelt sind, bietet sich so auch die Möglichkeit, dass die Oberfläche leicht ausgetauscht werden kann oder weitere Oberflächen angeboten werden können. Das könnte dann rein theoretisch eine Desktop-Anwendung oder eine eigene Android-App sein. Fehlende Redundanz der Daten bedeutet außerdem auch, dass Änderungen schnell an einer Stelle vorgenommen werden können und andere Vorkommen nicht vergessen werden können.

Die folgenden Daten müssen daher vom Server über die Schnittstelle bereitgestellt werden:

- Die Features, die generiert werden können.
- Die Kategorien, denen die Features zugeordnet sind.
- Die Presets.
- Die Syntaxen, in denen die Ontologie ausgegeben werden kann.
- Die generierte Ontologie mit einer langen URL.
- Die generierte Ontologie mit einer kurzen URL.

Natürlich muss auch eine Kommunikation zum Server hin möglich sein, über die die Ontologie generiert werden kann. Hierfür kommen die folgenden Daten in Frage:

- Die Features, die generiert werden sollen.
- Die Syntax, in der die generierte Ontologie ausgegeben werden soll.

Mit einer Webschnittstelle, die die genannten Daten verarbeiten und liefern kann, wäre die Weboberfläche, wie sie in Abschnitt 3.4 konzipiert wurde, voll funktionsfähig.

Generator

Die nächste und zentrale Komponente ist der Generator der Ontologien. Die grundlegenden Eingabedaten wurden bereits im vorigen Abschnitt aufgeführt: Die Features und die Syntax.

Initial sind nur die Features, die in der generierten Ontologie enthalten sein sollen, relevant. Der Generator soll dann aus den geforderten Features die Ontologie erstellen. Hierbei ist zu beachten, dass selbstverständlich alle geforderten Features berücksichtigt werden sollen, aber auch kein einziges Feature mehr. Würde die Ontologie umfangreicher als gefordert, könnte wegen mangelnder Unterstützung einzelner Elemente gegebenenfalls der gesamte Test der Visualisierung nicht durchgeführt werden.

Während der Entwicklung hat sich gezeigt, dass die Ontologie mit einer eigenen IRI identifiziert sein sollte. Da sich diese dann auch in der Ontologie wiederfindet, müsste sie eigentlich als zusätzlicher Eingabeparameter gewertet werden. Bei den Anforderungen an die Webschnittstelle ist dieser Parameter aber nicht aufgeführt, da die IRI nicht über die Schnittstelle geliefert werden kann. Eine korrekte, erreichbare IRI kann, ähnlich wie die lange oder kurze URL, nur vom Server generiert werden.

Im Anschluss an die Generierung soll die Ontologie mit der geforderten Syntax über die Webschnittstelle ausgegeben werden.

Datenbank

Im ursprünglichen Prozess war es nicht erforderlich, dass die Serveranwendung etwas speichern musste. Alle benötigten Daten konnten über die Webschnittstelle ausgetauscht werden, sodass der Server nie einen anderen Zustand hatte als bei Inbetriebnahme. Das änderte sich aber durch die Anforderung, eine kurze URL zu einer Ontologie ausgeben zu können.

In der ursprünglichen, langen URL waren alle Informationen erhalten, die der Server zur Generierung benötigte: Die Namen der Features waren hintereinander aufgelistet und auch die Syntax konnte angegeben werden. Theoretisch hätte man die Auflistung der Namen noch verkürzen können, was aber keine gute Lösung wäre, da man mit einer großen Feature-Anzahl trotzdem noch relativ lange URLs erhalten würde. Daher soll eine generierte Ontologie mit einer eindeutigen, kurzen Identifikationsnummer versehen werden, welche auch in der kurzen URL verwendet werden kann.

Eine Idee war, bei jeder Generierung einen Zähler um eins zu erhöhen und die Zahl dann als ID der Ontologie zu verwenden. Damit würden aber unnötig viele IDs angelegt, weil wiederholte Generierungen mit gleichen Eingabeparametern unterschiedliche IDs erhalten würden. Um die Anzahl der IDs gering und die IDs damit kurz zu halten, sollen äquivalente Ontologien die gleiche ID erhalten.

Der Server muss also die ID mit den Features der Ontologie verknüpfen und abspeichern, sodass er auch später anhand der kurzen URL in Erfahrung bringen kann, welche Features in der Ontologie enthalten sein sollen.

4. Implementierung

In diesem Kapitel wird die Umsetzung der Anforderungen und Ideen aus dem im vorigen Kapitel vorgestellten Konzept beschrieben. Eingeleitet wird mit einer Übersicht über die Technologien und der Umgebung, in der die Software entwickelt wird. Danach wird die Architektur der Anwendung gezeigt und deren Umsetzung anschließend genauer beschrieben.

4.1. Verwendete Technologien

Zu Beginn wird ein Überblick über die eingesetzten Technologien gegeben. Hierzu zählen die verwendeten Programmiersprachen, aber auch Software, die während des Entwicklungsprozesses genutzt wird, um diesen möglichst reibungslos ablaufen zu lassen.

4.1.1. Programmiersprachen

Durch die Anforderung an eine Webanwendung ist eine Hälfte der Technologien praktisch festgelegt: HTML, CSS und JavaScript werden für die Weboberfläche eingesetzt. Dabei soll das aktuellere JavaScript 5 verwendet werden, welches einige syntaktische Verbesserungen und bessere Grundlagen für eine Modularisierung mitbringt. Da JavaScript 5 schlecht von älteren Browsern unterstützt wird, muss es für die Verwendung im Browser mit *Babel*[1] zu JavaScript 4 kompiliert werden.

Im Gegensatz hierzu kam der Impuls für die Serverseite aus einer anderen Richtung. *OWL2VOWL*[2], eine Anwendung, die Ontologien in ein für die Ontologievisualisierung *WebVOWL* [LLMN14] interpretierbares Format konvertiert, habe ich während meiner Tätigkeit als studentische Hilfskraft mitentwickelt. Nach einer relativ kurzen Suche fiel die Entscheidung damals auf die Programmbibliothek *OWL API*[3], welche in [HB11] vorgestellt wird.

Eine weitere Programmbibliothek zum Arbeiten mit Ontologien ist *Apache Jena*. Wegen fehlender OWL 2-Unterstützung[4] kam diese aber im Vergleich zur wesentlich besser ausgestatteten OWL API nicht in Betracht.

Eine Programmbibliothek sollte auf jeden Fall verwendet werden, da es sich bei OWL um ein relativ komplexes Fachgebiet handelt. Die technische Unterstützung beim Arbeiten mit den einzelnen

[1] https://babeljs.io/
[2] https://github.com/VisualDataWeb/OWL2VOWL
[3] http://owlapi.sourceforge.net/<
[4] https://jena.apache.org/documentation/ontology/

OWL-Elementen, deren Kombinierung, den verschiedenen Syntaxen und den komplex aufgebauten OWL 2-Profilen kann den Aufwand wesentlich verringern.

Da die OWL API bereits bei OWL2VOWL gezeigt hat, dass sie sehr beim Erstellen und Verarbeiten von Ontologien helfen kann und außerdem auch aktiv weiterentwickelt wird, sollte sie hier verwendet werden. Da sie in der mir vertrauten Programmiersprache *Java* geschrieben ist, wird die serverseitige Anwendung ebenfalls in Java programmiert.

4.1.2. Build und Dependency-Management

Back-End

Als Buildsystem wird Apache Maven[5] verwendet, welches für die Entwicklung mit Java geeignet ist. Mit Maven werden in diesem Projekt die Quellcode-Dateien kompiliert, Tests ausgeführt und die kompilierten Dateien in einen lauffähigen `.jar`-Container gepackt.

Eine weitere Kernfunktion von Maven ist, die Programmbibliotheken in Form von Abhängigkeiten zu verwalten. Zu diesen Abhängigkeiten zählen zum Beispiel die OWL API, nützliche Bibliotheken, die die Entwicklung vereinfachen, oder auch das Framework, mit dem die Webanwendung realisiert wird. Maven sorgt dafür, dass jede Abhängigkeit in einer bestimmten Version vorliegt. Dadurch lässt sich auch schnell auf weiteren Systemen, wie beispielsweise dem eigenen Laptop, die Entwicklungsumgebung aufsetzen. Es ist dann sichergestellt, dass auch dort exakt die gleiche Anwendung erstellt wird.

Front-End

Da Maven nicht nativ mit Webtechnologien wie JavaScript zurecht kommt, wird dafür ein separates Buildsystem verwendet. Mit einem Plugin[6] für Maven lassen sich beide Buildsysteme kombinieren.

Zu den bekannteren Buildsystemen für Webtechnologien zählen *Grunt*[7] und *Gulp*[8], welche beide sehr ähnlich sind. Grunt wird bei der Ontologievisualisierung *WebVOWL* für ähnliche Aufgaben wie Maven in dieser Bachelorarbeit verwendet. Die Hauptfunktionalität von Grunt ist, bestimmte Aufgaben mit unterschiedlichsten Zielen – beispielsweise einzelne Schritte des Buildprozesses – auszuführen. Diese Aufgaben werden Grunt durch Plugins zur Verfügung gestellt und können anschließend in einer Konfigurationsdatei angepasst und kombiniert werden. Gulp unterscheidet sich zu Grunt hauptsächlich dadurch, dass temporäre Dateien im Build-Prozess in den Arbeitsspeicher gelegt werden, wodurch sich der Prozess beschleunigen kann.

Aus der mit Grunt gesammelten Erfahrung, ist die Konfiguration der Plugins häufig sehr zeitaufwendig, da jedes einzelne an die verwendeten Technologien und die Projektstruktur angepasst werden muss.

[5] https://maven.apache.org/
[6] https://github.com/eirslett/frontend-maven-plugin
[7] http://gruntjs.com/
[8] http://gulpjs.com/

Als Alternative hierzu stellt sich unter anderem *Brunch*[9] auf, auf welches ich erst kurz vor Beginn der Bachelorarbeit aufmerksam gemacht wurde. Brunch führt nicht einfach nur verschiedene Aufgaben aus Plugins aus, sondern implementiert die typischen Aufgaben für Webanwendungen bereits schon selbst und bietet hiermit einen Standardablauf an. Dazu zählen zum Beispiel das Konkatenieren von Dateien oder auch die Minimierung des Quellcodes, damit Webseiten schneller geladen werden können. Als Folge daraus ergeben sich ein minimaler Konfigurationsaufwand, aber auch etwas weniger Freiraum bei den Einstellungsmöglichkeiten. Da sich Brunch explizit als Alternative zu den beiden oben genannten Buildsystemen aufstellt, die Beispielprojekte sehr simpel aussahen und bei mir das Interesse geweckt war, wird Brunch hier eingesetzt.

Für clientseitige Webanwendungen hat sich die Paketverwaltung *bower*[10] zum Verwalten der Abhängigkeiten etabliert. Inzwischen findet auch die Paketverwaltung *npm* [11] von serverseitigen JavaScript-Anwendungen immer weitere Verbreitung bei clientseitigen JavaScript-Anwendungen. Leider entdeckte ich einen Bug[12] bei der Kombination von npm mit brunch und Windows, wodurch sie außer Betracht fiel.

4.1.3. Weiteres

Zur Versionsverwaltung wird ein Git-Repository verwendet. Im Vergleich zu SVN kann ich damit wesentlich flexibler arbeiten. Ein Beispiel wäre, dass man Änderungen in gebündelten Einheiten (sogenannte *Commits*) auch ohne eine verfügbare Internetverbindung speichern kann. In diesem Git-Repository sind der Quellcode und alle anderen Dokumente dieser Bachelorarbeit abgelegt. Das Repository wird auf Github gehostet, ist allerdings nicht öffentlich zugänglich.

Um den aktuellen Stand der Entwicklung selbst online testen zu können, ist OntoBench auf *Heroku*[13] installiert worden. Heroku ist ein *Platform-as-a-Service*-Dienstleister (PaaS), der unter anderem Server, eine Laufzeitumgebung oder eine Anbindung ans Internet anbietet. Auf so einer Plattform können Entwickler ihre Anwendungen mit minimalem Aufwand zum Laufen bringen und schnell der Öffentlichkeit zur Verfügung stellen, ohne sich Gedanken über technische Details wie beispielsweise die Infrastruktur machen zu müssen. Als PaaS-Dienstleister bietet Heroku außerdem auch eine Auswahl mehrerer Datenbanken an, welche in dieser Anwendung benötigt werden. In der kostenlosen Variante erhält man einen Server, der maximal 16 Stunden am Tag arbeiten darf, was für dieses Projekt aber ausreicht.

Heroku ist außerdem mit Github verbunden, sodass beim Einpflegen von Änderungen automatisch die aktuellste Version der Anwendung auf Heroku kompiliert und gestartet wird. Da die Anwendung in einer komplett neu eingerichteten Umgebung gestartet wird, ist gewährleistet, dass keine Altlasten der Software eine neuere Version stören. Dadurch ist für mich als Entwickler sichergestellt, dass die Anwendung nicht nur auf meinem lokalen System lauffähig ist. Für meine Betreuer hat es außerdem den Vorteil, dass sie zu jedem Zeitpunkt den aktuellen Stand testen können.

[9] http://brunch.io/
[10] http://bower.io/
[11] https://www.npmjs.com/
[12] https://github.com/brunch/brunch/issues/1009
[13] https://www.heroku.com/

Programmiert wird in der Entwicklungsumgebung IntelliJ IDEA[14] von JetBrains. Im Vergleich zum bekannten Eclipse kann ich mit IntelliJ wesentlich effektiver arbeiten, was zum Beispiel von besseren kontextsensitiven Vorschlägen bei der Autovervollständigung herrührt. Außerdem werden Windows und Linux als Betriebssysteme verwendet, wodurch auch wieder eine gewisse Grundkompatibilität gewährleistet ist.

4.2. Entwurf

Im Konzept wurde bereits festgelegt, dass eine Webanwendung entwickelt werden soll. Damit ergibt sich bereits ein bedeutender Aspekt der Architektur: Die Trennung zwischen Server und Client. Der zweite Aspekt äußert sich durch die Forderung nach dem Speichern von IDs in Verbindung mit dem Inhalt der Ontologie in einer Datenbank.

Wenn man nun die gesamte Anwendung betrachtet, ergeben sich für mich die folgenden, relativ gut in sich abgeschlossenen Komponenten, welche in Abb. 4.1 abgebildet sind. Die Pfeile zwischen den einzelnen Komponenten, dem Internet und dem Anwender stehen dabei für den Fluss der Informationen. In Abbildungen der Architektur steht *blau* für die Clientseite und *orange* für die Serverseite.

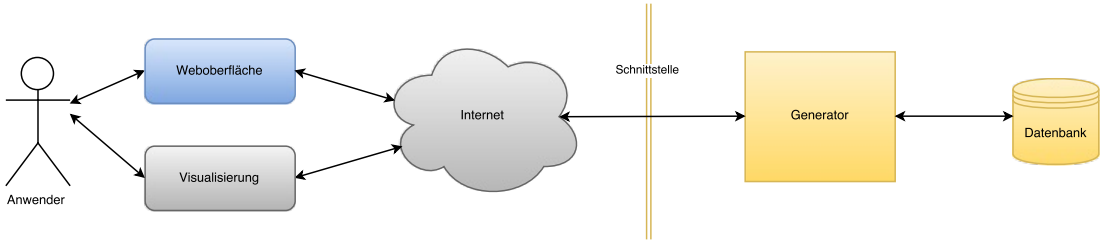

Abbildung 4.1.: Eine Übersicht über die groben Komponenten der gesamten Anwendung und deren Informationsfluss. Die eingefärbten Komponenten sind Teil von OntoBench.

4.2.1. Weboberfläche

Die Weboberfläche ist der Teil der Anwendung, der direkt vom Anwender angesprochen wird. Hier findet die Anzeige und Auswahl der Features und Presets statt, aus denen der Anwender die Ontologie generieren kann. Zusätzlich gibt es auch die Möglichkeit, eine spezielle Syntax für die Ontologie wählen zu können. Die Weboberfläche soll keinerlei Informationen über die verfügbaren Features, Kategorien, Presets und Syntaxen speichern, denn diese sollen vom Server zentral bereitgestellt werden.

Es soll noch einmal angemerkt sein, dass sich die Laufzeitumgebungen vom Server und der Weboberfläche unterscheiden. Zwar liegen die Dateien von beiden auf einem Server, allerdings wird die Weboberfläche in den Browsers des Benutzers geladen und dann dort ausgeführt. Um mit dem

[14] https://www.jetbrains.com/idea/

Server zu kommunizieren, ist deshalb eine Webschnittstelle nötig, die über das Internet angesprochen werden kann.

4.2.2. Webschnittstelle

Die Schnittstelle steht zwischen der Weboberfläche und dem Server. Über die Schnittstelle werden Daten in beide Richtungen ausgetauscht; eine genaue Auflistung dieser befindet sich im Konzept in Abschnitt 3.5.1. Die Schnittstelle hat keinen eigenen Zustand und reicht nur Daten zwischen den beiden umliegenden Komponenten durch.

Abgesehen davon, kann eine Ontologie auch über ihre eindeutige URL abgerufen werden, wobei die Weboberfläche übergangen wird. Nach meiner Vorstellung tritt der Fall dann ein, wenn die Ontologie zum Validieren einer Visualisierung eingelesen und verarbeitet wird; prinzipiell sollen aber auch andere Szenarien nicht ausgeschlossen werden.

4.2.3. Generator

Der Generator stellt praktisch das Herzstück der Anwendung dar und enthält den größten Teil der Logik. Hier werden die Daten von der Webschnittstelle in Empfang genommen und ausgewertet. Anschließend wird daraus die Ontologie erzeugt und über die Webschnittstelle zurückgegeben.

Um eine Ontologie aus einer kurzen ID generieren zu können, müssen die beinhalteten Features mitsamt der ID abgespeichert werden, wofür eine Datenbank erforderlich ist.

4.2.4. Datenbank

Die Datenbank enthält grundsätzlich keine Logik; sie dient ausschließlich als persistenter Speicherort. Die zu speichernden Daten bestehen aus einer ID in Verbindung mit den zugehörigen Features.

4.3. Webschnittstelle – Spezifikation

Wie oben beschrieben, steht zwischen der Serveranwendung und der clientseitigen Weboberfläche eine Schnittstelle. Da sie zentral für die Planung der beiden umgebenden Komponenten ist, soll sie hier auch als erstes spezifiziert werden. Auf die Beschreibung der Technologien folgt die genaue Spezifikation der Schnittstelle.

4.3.1. Technologien

Die Schnittstelle soll das *REST*-Architekturmuster[15] implementieren, mit welchem die hier verwendeten Technologien einfach miteinander kommunizieren können. Über HTTP werden Daten zwischen den Kommunikationspartnern ausgetauscht. Der gesamte Zustand der Kommunikation befindet sich dabei ausschließlich in den Nachrichten. Bei einer Client-Server-Architektur wie in dieser Anwendung, ergeben sich dadurch einige Vorteile. Wenn viele Anwender die Software gleichzeitig bedienen, werden viele Anfragen an den Server gesendet. Da der Server keinerlei Informationen zu den Anfragen der Anwender sammelt, wird nicht direkt zusätzlicher Speicher benötigt, um allein die Kommunikation bei vielen Anfragen instand zu halten. Die Kommunikationspartner sind dadurch gut voneinander entkoppelt. Außerdem wird REST beziehungsweise die Kommunikation über HTTP von JavaScript aus bereits gut unterstützt.

Die kommunizierten Daten könnten dabei als Reintext übermittelt werden. Um aber eine gewisse Struktur zu verwenden, wird meistens auf XML oder JSON als Datenformat gesetzt. JavaScript kann ohne Weiteres JSON, ausgeschrieben *JavaScript Object Notification*, interpretieren und verarbeiten. Da JSON außerdem kompakter und meiner Meinung nach lesbarer als XML ist, wird es hier als Datenformat verwendet.

4.3.2. Features

Von der Schnittstelle erwartet die Weboberfläche eine Liste der verfügbaren Features. Die einzelnen Features können auf die folgenden drei Eigenschaften reduziert werden:

Name Der Name soll entweder das dahinterstehende OWL-Element identifizieren oder allgemein das Feature beschreiben.

Kategorie-ID Die Features sollen für eine bessere Übersicht gruppiert werden, wofür die zugehörige Kategorie benötigt wird. Wie im folgenden Abschnitt beschrieben ist, wird hier nur die ID einer Kategorie benötigt.

Token Eigentlich sollte ein Name eines Features bereits eindeutig sein. Um ein Feature aber kompakt und eindeutig identifizieren zu können, soll ein kurzes Token verwendet werden.

Eventuell könnten auch eine etwas längere Beschreibung und weitere Informationen wie beispielsweise die zugehörigen OWL-Profile übertragen werden. Um den Aufwand aber gering zu halten, werden sprechende, kurze Namen für die einzelnen Features gewählt.

[15] http://www.ibm.com/developerworks/library/ws-restful/

4.3.3. Kategorien

Da mehrere Features in einer Kategorie liegen, würde man eine große Redundanz erhalten, wenn jedes einzelne Feature alle Informationen zu seiner Kategorie enthält. Deshalb sollen die Kategorien getrennt von den Features über die Schnittstelle angeboten werden.

Ich habe mich dafür entschieden, die Kategorie eines Features diesem fest zuzuordnen. Dadurch ist gewährleistet, dass jedes Feature auch nur exakt einer Kategorie zugeordnet sein kann. Die andere, ähnliche Möglichkeit wäre gewesen, die Kategorie als Container zu betrachten, in dem die einzelnen Features liegen; der Unterschied zur vorigen Variante ist aber nicht maßgeblich.

Die Schnittstelle bietet eine Liste der Kategorien an, welche die einzelnen Kategorien wie folgt beschreibt:

ID Die ID, mit der die Kategorie bei den Features referenziert wird.

Name Der Name der Kategorie.

Index Durch den Index können alle Kategorien untereinander sortiert werden.

4.3.4. Presets

Im Gegensatz zu den Kategorien kann ein Feature mehreren Presets zugeordnet sein. Daher habe ich die Beziehung von Features zu den Presets umgedreht, wie es alternativ bei den Kategorien möglich gewesen wäre. Ein Preset ist also ein Container, der eine Menge von Features enthält.

Wie bei den Abschnitten zuvor, wird auch hier wieder eine Liste angeboten. Die einzelnen Presets darin sind wie folgt konstruiert:

Name Der Name des Presets.

Features Eine Auflistung der Features. Hier sollen nur die Token und nicht die ganzen Features enthalten sein.

Index Wie bei den Kategorien macht es auch hier Sinn, die einzelnen Presets sortieren zu können. Als Beispiel könnten die aufeinander aufbauenden OWL 1-Profile geordnet aufgelistet werden.

4.3.5. Syntaxen

Aus der Liste der verfügbaren Syntaxen kann der Anwender in der Oberfläche wählen. Die getroffene Auswahl wird anschließend zum Generieren der Ontologie verwendet.

Name Der Name der Syntax.

Dateiendung Die Dateiendung des Datenformats einer Syntax. Sie soll wie in Abschnitt 3.2.8 in der URL angegeben werden.

Standardauswahl Eine Angabe, ob diese Syntax standardmäßig gewählt werden soll.

4.3.6. Ontologie

Im Gegensatz zu den vorigen Methoden der Schnittstelle nimmt der Server zum Generieren der Ontologie Daten entgegen. Hierbei soll die URL als einzige Informationsquelle dienen.

In der langen URL wird der Inhalt der Ontologie durch eine Auflistung der Feature-Tokens angegeben, in der kurzen URL nur durch die ID. Beide URLs haben gemeinsam, dass ähnlich wie `.html` bei vielen Webseiten die Dateiendung der Ontologie die Syntax angeben sollen, wie es auch in Abschnitt 3.2.8 abgebildet ist. Als Antwort soll dann direkt die Ontologie zurückgegeben werden.

4.4. Weboberfläche

Die Weboberfläche basiert fast vollständig auf den über die Schnittstelle bereitgestellten Daten. Sie wird dynamisch aus ihnen generiert und ist deshalb sehr flexibel aufgebaut. Da ihre Funktionalität sehr überschaubar ist, wird die Architektur nur relativ knapp beschrieben. Anschließend wird ein ausführlicher Überblick über die einzelnen Komponenten der Oberfläche gegeben.

4.4.1. Entwurf

Die Weboberfläche soll aus einer einzigen Webseite bestehen. Auf dieser Webseite kann der Nutzer zwischen Tabs für die Auswahl der Features und die Generierung und Ausgabe der Ontologie wechseln.

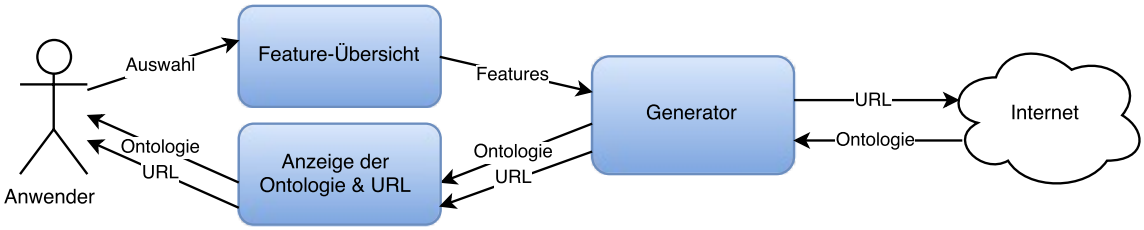

Abbildung 4.2.: Die Architektur der Weboberfläche mitsamt des Datenflusses.

Die Oberfläche der Webseite erhält ihre Struktur durch HTML und wird mit CSS gestaltet. Mit JavaScript werden die dynamischen Inhalte in die Oberfläche eingebaut, Feedback zu Benutzerinteraktionen gegeben und die Webschnittstelle über das Internet angesprochen.

4.4.2. GUI Framework

Um schnell eine einheitliche und hübsche Weboberfläche aufzubauen, verwende ich das GUI-Framework *Semantic UI*[16]. Mit Semantic UI werden Webseiten mit natürlichsprachlichen Bezeichnern

[16] http://semantic-ui.com/

gestaltet. Da diese sehr intuitiv gewählt sind, können unbekannte Bezeichner oft erraten werden. Außerdem bleibt der Quellcode hierbei sehr lesbar, da er ziemlich selbst beschreibend ist. Semantic UI erweitert nicht nur bereits von Browsern unterstützte HTML-Elemente wie Checkboxen, sondern bietet auch eine Vielzahl neuer Komponenten wie beispielsweise eine Fortschrittsanzeige an.

In Listing 4.1 ist ein Auszug aus dem Code zu sehen, der die Übersicht der Features beschreibt. Die `div`-Elemente stellen reine Container dar, die selbst keine Funktion besitzen. Im äußeren Element wird mit den CSS-Klassen ein zweispaltiges Gitternetz aufgespannt. Darin liegen die Container für die Kategorien der Features. Um nun ein einspaltiges Gitter zu erhalten, würde man bloß *two columns* durch *one column* ersetzen.

Listing 4.1 Definition eines Gitternetzes mit Semantic UI.

```
<div class="ui two columns grid">
      <div class="ui column">...</div>
      <div class="ui column">...</div>
</div>
```

Semantic UI ist eine Erweiterung von *jQuery*[17]. In JavaScript kann mit jQuery leicht der HTML-Code manipuliert werden oder auch die Logik für Benutzerinteraktionen implementiert werden. jQuery bietet außerdem eine sehr gut abstrahierende Schnittstelle zum Internet an, die in OntoBench für die Kommunikation mit dem Server verwendet wird.

4.4.3. Module

Der JavaScript-Quellcode für die Weboberfläche ist in drei Module aufgeteilt. Das erste Modul enthält allgemeine Funktionen, die für die gesamte Webseite verwendet werden. Von hier aus werden alle Komponenten initialisiert, die dynamischen Daten vom Server abgefragt und diese anschließend in die Oberfläche geladen. Zu diesen Komponenten gehören die mit Semantic UI realisierten Tabs oder auch die Presets.

Im nächsten Modul befindet sich die Logik der Feature-Übersicht. Hier wird das Gitternetz mit den Kategorien und deren Features konstruiert und in die Oberfläche geladen.

Die Generator-Oberfläche liegt als weiterer großer Teil ebenfalls in einem eigenen Modul. In diesem wird aus den ausgewählten Features die URL für die Ontologie erstellt, die Anfrage zum Server gesendet und anschließend die erhaltene Ontologie mitsamt der jeweiligen URL dargestellt.

4.4.4. Komponenten

Die Weboberfläche ist aus mehreren, kleineren Komponenten aufgebaut. Im HTML-Code werden diese angeordnet, sodass die Struktur der Oberfläche festgelegt wird.

[17] https://jquery.com/

Quick Guide

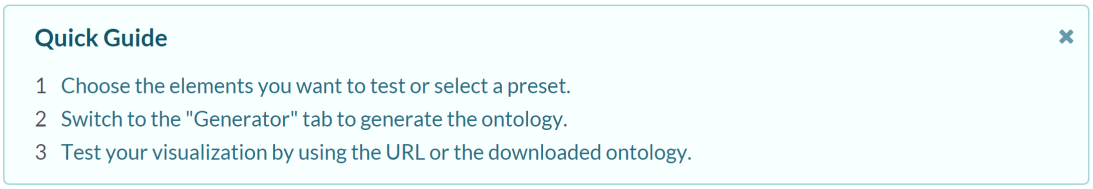

Abbildung 4.3.: Zu Beginn wird dem Anwender die grundlegende Funktionalität erklärt.

Um dem Anwender zu erklären, wie er eine Ontologie generieren kann, werden die benötigten Schritte in Kürze beschrieben. Diese Kurzanleitung ist im oberen Teil der Webseite über den Tabs platziert, damit sie auch beim Wechsel zwischen diesen sichtbar ist. Sie hebt sich außerdem auch gegenüber anderer Elemente farblich hervor.

Der Quick Guide wird meiner Meinung nach nur zu Beginn als Einstiegshilfe benötigt; im weiteren Zeitverlauf belegt er unnötigerweise Platz in der Oberfläche. Daher kann er mit einem Klick auf die in Abb. 4.3 oben rechts dargestellte Schaltfläche dauerhaft geschlossen werden.

Beim Anwender wird dabei lokal ein sogenannter *Cookie* gespeichert. Ein Cookie kann mit einen Namen, Textinhalt und einer Gültigkeitsdauer vom JavaScript-Code aus erstellt und gelesen werden. Wird die Kurzanleitung geschlossen, wird ein Cookie mit einer unbegrenzten Gültigkeitsdauer erstellt. Sobald der Nutzer die Webseite erneut aufruft, wird die Existenz des Cookies geprüft und gegebenenfalls die Anleitung ausgeblendet.

Tabs

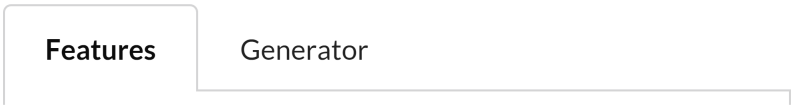

Abbildung 4.4.: Um die Weboberfläche übersichtlich zu halten, wird der Inhalt auf zwei separaten Ansichten dargestellt. Über Tabs kann zwischen diesen gewechselt werden.

Die Feature-Übersicht und der Generator sind in unterschiedlichen Ansichten platziert. Über die Tabs aus Abb. 4.4 kann der Anwender zwischen diesen wechseln. Sobald man den Generator-Tab öffnet, wird im Hintergrund die Generierung der Ontologie angestoßen.

Presets

Im oberen Bereich des Feature-Tabs sind die Presets besonders hervorgehoben, da sie für den Anwender eine große Zeitersparnis darstellen können. Um zu signalisieren, dass es sich um keine Schalter handelt, die die Features der Presets an- beziehungsweise abwählen, ist dem Namen jedes Presets ein Plus-Icon vorangestellt. Der Nutzer soll hieraus erkennen, dass die Features ausschließlich zur Auswahl hinzugefügt werden.

Abbildung 4.5.: Die Presets werden besonders hervorgehoben, da sie dem Anwender die Auswahl der Features wesentlich erleichtern.

Beim Aufruf der Weboberfläche werden die verfügbaren Presets vom Server abgefragt und anschließend die einzelnen Schaltflächen erstellt. Semantic UI sorgt hierbei dafür, dass trotz dynamischer Anzahl alle GUI-Elemente die gleiche Größe besitzen. Da die Schaltflächen bei niedrigeren Auflösungen zu klein sind, werden sie mit Hilfe von Semantic UI nicht mehr nebeneinander sondern untereinander platziert.

Features

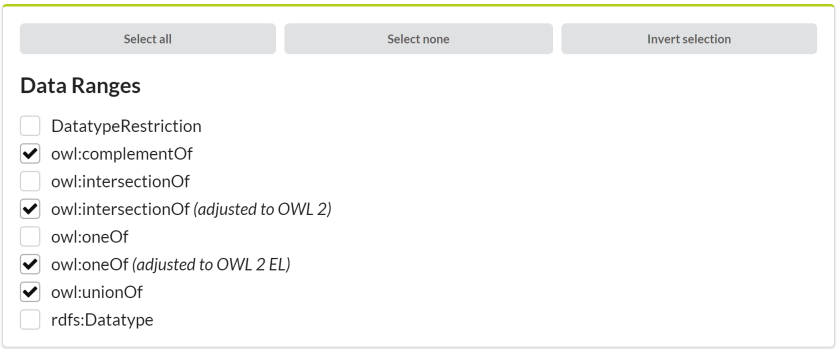

Abbildung 4.6.: Die Features einer Kategorie liegen jeweils in einem eigenen Container.

Die Features werden nach ihrer Kategorie gruppiert und anschließend in abgeschlossenen Containern wie in Abb. 4.6 angezeigt. Diese Container werden in der Feature-Übersicht in einem zweispaltigen Gitter angeordnet.

In jedem Container sind oben drei Schaltflächen platziert, die gröbere Auswahlmöglichkeiten bieten. So kann man pro Container alle Features an- oder abwählen und die aktuelle Auswahl invertieren, was vor allem bei größeren Containern praktisch ist. Die gleichen Schaltflächen liegen auch nochmal über allen Containern, um die Features aller Kategorien gleichzeitig manipulieren zu können.

Darunter liegt der eigentliche Inhalt: Der Name der Kategorie und die einzelnen Features. Hinter dem Namen eines Features werden gegebenenfalls erläuternde Informationen angezeigt, wenn beispielsweise ein Feature explizit für ein OWL Profil konstruiert ist. Der Anwender kann die Features mit einem einfachen Klick zur Auswahl hinzufügen beziehungsweise von dieser entfernen.

Generator

Der zweite Tab enthält Komponenten, die den Generator und die Ontologie betreffen. Über das in Abb. 4.7 links abgebildete Dropdown-Menu kann die Syntax der Ontologie eingestellt werden. Direkt nach einer Änderung wird wie beim Wechseln der Tabs die Ontologie erneut generiert. In der Mitte

Abbildung 4.7.: Die GUI-Komponenten, mit denen der Generator konfiguriert und die Ontologie genutzt werden kann.

der Abbildung wird die URL der Ontologie angezeigt. Über einen Schalter kann hier zwischen der langen und der kurzen URL gewechselt werden. Sobald der Anwender das Feld fokussiert, wird die URL markiert und ist somit direkt kopierbar. Mit einem Klick auf den Download-Button auf der rechten Seite kann die Ontologie heruntergeladen werden.

Unter den oben beschriebenen Elementen wird die generierte Ontologie angezeigt. Sie wird wie die URL markiert, sobald das Feld im Fokus liegt und kann dadurch auch ohne einen Download extrahiert werden.

```
@prefix : <http://localhost:8080/ontology/1/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://localhost:8080/ontology/1/> .

<http://localhost:8080/ontology/1/> rdf:type owl:Ontology .

###  Generated by the OWL API (version 4.0.2.20150417-2043) http://owlapi.sourceforge.net
```

Abbildung 4.8.: Die Anzeige der generierten Ontologie, welche in diesem Fall keine Features enthält.

4.5. Server

Im Gegensatz zur Weboberfläche ist die auf dem Server laufende Anwendung wesentlich komplexer. Hier liegen neben den Daten für die Oberfläche die komplette Logik und ein Datenspeicher für den Inhalt bereits generierter Ontologien.

4.5.1. Architektur

Im Entwurf in Abb. 4.1 haben sich bereits die groben Komponenten gezeigt, die der Weboberfläche auf der Serverseite gegenüberstehen. Auf die Implementierung bezogen, haben sich daraus die in Abb. 4.9 abgebildeten Komponenten ergeben.

Die Webschnittstelle stellt entweder Daten bereit, wie beispielsweise die verfügbaren Features, oder nimmt Parameter zum Generieren einer Ontologie an. Diese Parameter können entweder eine direkte Auflistung der Features oder eine ID sein, welche erst in die dahinter stehenden Features aufgeschlüsselt werden muss. Daraufhin kann der Generator angesprochen und die resultierende Ontologie über die Schnittstelle zurückgegeben werden.

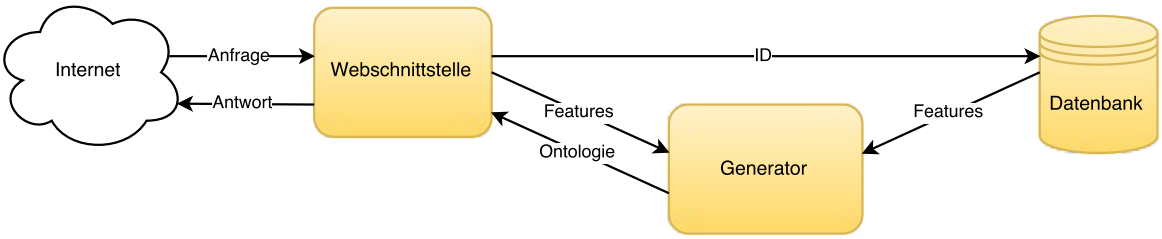

Abbildung 4.9.: Die Architektur der serverseitigen Anwendung.

4.5.2. Framework

Durch *Spring Boot*[18], das auf dem *Spring Framework*[19] aufbaut, kann schnell eine Webanwendung mit REST-Schnittstelle implementiert werden. Mit Spring Boot sollen Entwickler ohne eine aufwendige Konfiguration vieler separater Bibliotheken schnell eine startbare Anwendung bauen können. Hierfür werden übliche Anwendungsfälle wie eine Webschnittstelle oder die Kommunikation mit einer Datenbank von Spring Boot gekapselt und vorkonfiguriert. Anstatt viele einzelne Bibliotheken erst an die Anwendung anpassen zu müssen, kann direkt ihre Funktionalität genutzt werden. Da die vorkonfigurierten Einstellungen von Spring Boot allerdings nicht für jedes Anwendungsgebiet passen, können auch noch Feinjustierungen erledigt werden.

Abgesehen von einer sehr guten Abstrahierung der Implementierung einer Webschnittstelle oder der Datenbankkommunikation kann mit Spring *Dependency Injection* sehr flexibel eingesetzt werden. Unter Dependency Injection versteht man, dass eine Klasse nicht selbst Objekte anderer Klassen instantiiert, sondern sie diese von außen beispielsweise über den Konstruktor erhält. Dadurch werden einzelne Module einer Anwendung besser voneinander entkoppelt.

Auf diese Anwendung bezogen, könnte beispielsweise das Generieren der Ontologie aus einer URL ohne einen funktionsfähigen Generator getestet werden. Hierfür müsste ein sogenannter *Stub* des Generators implementiert werden. Dieser Stub liefert vordefinierte Antworten, in diesem Fall immer dieselbe Ontologie. Wird nun eine Anfrage an die bereits implementierte Webschnittstelle simuliert, fragt diese beim Generator nach der Ontologie und gibt sie anschließend an die simulierte Anfrage zurück. Da der Generator von außen in die Webschnittstelle gegeben wurde, war dieser zu keinem Zeitpunkt bewusst, dass es sich nicht um einen voll funktionsfähigen Generator, sondern um einen Stub handelt. Zusammengefasst wäre dadurch bestätigt, dass die implementierte Webschnittstelle korrekt funktioniert. Nun könnte der Fokus auf die Implementierung des Generators gelegt werden.

4.5.3. Webschnittstelle

Zu Beginn der Entwicklung lag der Fokus stark auf der Webschnittstelle. Sobald diese funktionsfähig war, konnte nämlich auch parallel mit der Implementierung der Weboberfläche begonnen werden.

[18] http://projects.spring.io/spring-boot/
[19] http://spring.io/

Dadurch konnte ich einzelne Anforderungen an die gesamte Anwendung bereits frühzeitig in sogenannten *End-to-End*-Tests testen. Eine dieser Anforderungen war beispielsweise das Anzeigen einer Kategorie mitsamt der zugehörigen Features.

Zuordnung der URLs

Damit Spring erkennt, dass eine Webschnittstelle konfiguriert werden soll, muss eine Java-Klasse mit `@RestController` annotiert werden. In dieser Klasse werden dann Methoden beschrieben, die die Funktionalität der Schnittstelle implementieren.

Eine Methode wird wie in Listing 4.2 auf eine URL gemappt. In der Annotation `@RequestMapping` wird beschrieben, dass es sich um den Pfad `/ontology/` mit dem Parameter `features` handelt. Spring verarbeitet diese Annotation beim Start der Anwendung und verknüpft sie mit dem Pfad der Webschnittstelle. In diesem Fall wäre `/ontology/?features=owlthing,owlclass` eine gültige Adresse.

Listing 4.2 Verknüpfen der Schnittstellen-URL mit einer Java-Methode.

```
@RequestMapping(value = "/ontology/", params = "features")
public OWLOntology ontology(@RequestParam("features") List<Feature> features) {
  // generate and return the ontology
}
```

Spring verknüpft aber nicht nur URLs mit einer bestimmten Java-Methode, sondern kann auch die textuellen Parameter in Java-Objekte konvertieren.

Konvertieren von Werten

An der Schnittstelle findet die Konvertierung von Parametern der URL in Java-Objekte und umgekehrt von Java-Objekten in eine Textform statt.

Beim Einlesen der URL kann Spring mit einem speziellen Konverter Parameter der URL in einen bestimmten Java-Typen umwandeln. In Listing 4.2 wird beispielsweise eine Auflistung von Tokens in eine Liste von Feature-Objekten gebracht. Für jede benötigte Konvertierung der Parameter ist deshalb ein eigener Konverter implementiert.

Für die Ausgabe von Daten über die Schnittstelle müssen diese laut der Schnittstellenbeschreibung entweder in JSON oder in einer Syntax für Ontologien vorliegen. Die Konvertierung von Objekten in das JSON-Format ist mit Spring trivial, da es diese Funktionalität bereits standardmäßig unterstützt. Im Gegensatz hierzu ist die Ausgabe der Ontologie in einer speziellen Syntax komplizierter.

Die Anwendung muss zuerst erkennen, welche Syntax angefordert wird. Spring bietet hierfür eine Schnittstelle, mit der mit Hilfe einer Mustererkennung die Dateiendung der Syntax aus der URL ausgelesen wird. Anschließend wird mit einem Konverter die Ontologie durch die OWL API umgewandelt und ausgegeben.

4.5.4. Generator

Der Generator ist generisch implementiert und besitzt selbst keine fachliche Funktionalität. Seine Eingabeparameter sind eine leere Ontologie und die Objekte der zu berücksichtigenden Features, die er zum Beispiel von der Schnittstelle erhält. In diesen Objekten ist der Code gekapselt, mit dem die Ontologie durch die OWL API beim Generieren gefüllt wird.

Aufbau eines Features

Jedes Feature ist in einer separaten Klasse implementiert und von anderen Features entkoppelt. Die einzelnen Features sind alle analog zum Code-Ausschnitt aus Listing 4.3 aufgebaut.

Listing 4.3 Die Implementierung des `owl:Thing` Features (vereinfacht).

```
public class OWLThingFeature {
  public void addToOntology(OWLOntology ontology) {
    // add OWL elements to ontology
  }

  public String getName() {
    return "owl:Thing";
  }

  public String getToken() {
    return "thing";
  }

  public FeatureCategory getCategory() {
    return FeatureCategory.PREDEFINED_CLASSES;
  }
}
```

Die Methode `addToOntology` stellt dabei das Herzstück jedes Features dar. In dieser Methode werden über die OWL API die entsprechende Struktur von OWL-Elementen konstruiert und anschließend zur Ontologie hinzugefügt. Beim Generieren ruft der Generator diese Methode zusammen mit der Ontologie, die zu diesem Zeitpunkt erstellt wird, auf.

Im unteren Bereich der Klasse wird sozusagen der Steckbrief eines Features erstellt. Für jedes Feature ist hier der Name und sein eindeutiges Token angegeben. Es wird ebenso auf die zugehörige Kategorie verwiesen.

Beim Start der Anwendung tragen sich alle verfügbaren Features in einer zentralen Registrierung ein. Um mögliche Probleme aufgrund mehrfach vorkommender Tokens oder Namen zu vermeiden, findet hier eine Validierung dieser Attribute statt. Sollte sich ein Duplikat eingeschlichen haben, wird der Start der Anwendung mit einem Verweis darauf verweigert. Die Registrierung wird außerdem auch von der Webschnittstelle angesprochen, wenn alle verfügbaren Features abgerufen werden.

4.5.5. Datenbank

Um eine ID mit einer unbestimmten Menge an Features zu verknüpfen, werden diese in einer relationalen Datenbank abgelegt.

Initialisierung der Datenbank

In der Anwendung verwende ich standardmäßig die *HyperSQL*[20]-Datenbank (auch *HSQLDB* genannt), welche über Maven als Abhängigkeit eingebunden ist. Für die Entwicklung bietet diese Datenbank einen enormen Vorteil, da sie im *memory-only*-Modus mit der Anwendung gestartet werden kann. Das heißt, dass die Datenbank nur im Arbeitsspeicher existiert und auch nur, solange die Anwendung läuft. Beim Start der Anwendung liegt somit immer eine leere Datenbank vor, wodurch keine Altlasten aus vorherigen Entwicklungsschritten vorhanden sind.

Um eine Datenbank von Java aus zu verwenden, muss zuerst eine Verbindung zu dieser aufgebaut werden. Spring ist unter anderem für HSQLDB vorkonfiguriert und kann dadurch automatisch die Verbindung zur Datenbank herstellen, insofern diese im memory-only-Modus gestartet wird.

Im Produktiveinsatz soll die Datenbank die Daten allerdings dauerhaft und nicht nur für die Laufzeit der Anwendung speichern. In diesem Fall verwendet man eine Datenbank, die separat gestartet wird. Mit *Heroku Postgres*[21] bietet Heroku eine solche Datenbank an.

Auf Heroku kann die Anwendung mit Heroku Postgres verbunden werden. Heroku stellt die Adresse der Datenbank und die Zugangsdaten über eine Umgebungsvariable des Systems bereit. Wird die Anwendung gestartet, muss zwischen der externen Heroku Postgres-Datenbank oder der HyperSQL-Datenbank im memory-only-Modus entschieden werden. An dieser Stelle wird Spring durch Überprüfen der Umgebungsvariablen mitgeteilt, welche Datenbank verwendet werden soll und gegebenenfalls welche Zugangsdaten zu dieser Datenbank gehören.

Entity-Relation-Modell der Datenbank

Die Daten, die für das Speichern einer Generierung benötigt werden, sind wie folgt aufgebaut: Zum einen gibt es die Generierung, zum anderen die einzelnen Features. Jede Generierung ist mit einer unbestimmte Menge an Features verknüpft.

Wird diese Datenstruktur auf eine Datenbank abgebildet, ergibt sich das in Abb. 4.10 gezeigte *Entity-Relation*-Modell. Die Entität Generierung (englisch *Generation*) wird über ihre ID eindeutig identifiziert. Die Entität Feature wird durch das jeweilige Token eindeutig identifiziert.

Verknüpft werden beide Entitäten in einer sogenannten Junction-Table, die in der Abbildung blau dargestellt ist. Diese Tabelle enthält einzelne Parameter, die eine Generierung und ein Feature referenzieren.

[20] http://hsqldb.org/
[21] https://www.heroku.com/postgres

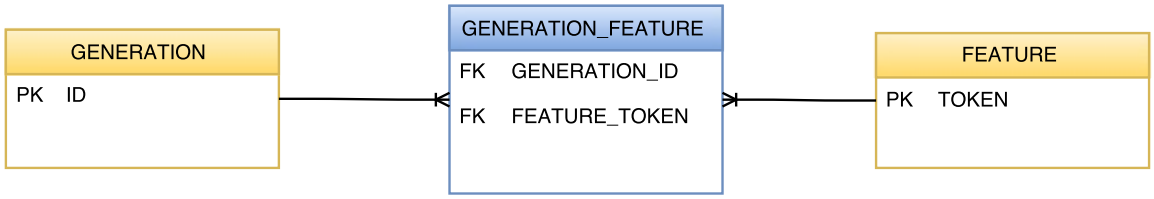

Abbildung 4.10.: Das Entity-Relation-Modell der Datenbank.

Im jetzigen Zustand sind die einzelnen Tabellen noch relativ leer. In Zukunft könnten allerdings noch weitere Informationen wie beispielsweise das Datum der Generierung gespeichert werden. Eine andere, wichtige Idee für eine Erweiterung ist in Abschnitt 6.2 weiter ausgeführt.

Speichern und Lesen der Java-Objekte

Die *Java Persistence API*, kurz JPA, spezifiziert, wie Klassen durch Java-Annotationen auf ein Datenbankmodell abgebildet werden können. Die Annotationen enthalten beispielsweise Tabellen- oder Spaltennamen.

Spring bringt die Anwendungsbibliothek *Hibernate*[22] mit sich, welche eine funktionsfähige Implementierung der JPA darstellt. Hibernate verarbeitet die Annotationen und erstellt daraufhin im Hintergrund mit SQL-Befehlen die erforderliche Tabellenstruktur.

Zum Speichern und Lesen der gespeicherten Daten wird ein sogenanntes *Repository* von Spring verwendet. In einem Repository kapselt Spring die Hibernate-Bibliothek, um die Schnittstelle zur Datenbank noch weiter zu abstrahieren. So werden ohne eine einzige Zeile SQL alle Datenbankoperationen vom Java-Code aus gesteuert.

[22] http://hibernate.org/orm/

5. Evaluation

Die Ziele dieser Bachelorarbeit können aus unterschiedlichen Sichtweisen definiert werden. In diesem Kapitel wird zum einen die entwickelte Anwendung an sich und die generierte Ontologie überprüft. Zum anderen wird getestet, inwiefern die Funktionalität einer Ontologievisualisierung mit dieser Anwendung validiert werden kann.

5.1. Funktionalität der Anwendung

Damit die Anwendung selbst als lauffähig bezeichnet werden kann, muss sie in einem praxisnahen Umfeld mit einer entsprechenden Umgebung funktionieren. Außerdem muss auch bestätigt werden, dass sie die geforderte Funktionalität leistet.

5.1.1. Umgebungsbedingungen

Die Anwendung kann auf unterschiedlichen Betriebssystemen betrieben und genutzt werden. Auf der Serverseite wird das unter anderem durch Verwenden von Java als Programmiersprache erreicht. Eine Java-Anwendung wird nämlich in einer sogenannten *Java Virtual Machine*, kurz *JVM* gestartet. Die JVM bietet der Java-Anwendung eine vordefinierte Laufzeitumgebung, in der Eigenschaften der Systemumgebung gekapselt werden.

Dieser Aspekt wurde zum einen dadurch getestet, dass die Anwendung auf Heroku parallel zur Entwicklung und auch danach für weitere Tests intern genutzt wurde. Heroku bietet unter anderem verschiedene Anwendungsserver in der Cloud, die sich hauptsächlich in der verfügbaren Leistung unterscheiden. In diesem Fall wurde zwar nur die kostengünstigste Variante gewählt, welche allerdings trotzdem eine praxisnahe Laufzeitumgebung bietet.

Zum anderen wurde auch auf einem Laptop mit der Linux-Distribution *Arch Linux* und auf einem PC mit *Windows 8* und *Windows 10* entwickelt und getestet, wodurch dieser Gesichtspunkt in gewissem Maß auch nochmal bestätigt wird.

Die Weboberfläche, also der clientseitige Teil der Anwendung, ist ebenfalls in unterschiedlichen Browsern lauffähig. Das wurde einerseits dadurch erreicht, dass die Anwendungsbibliothek jQuery[1] browserspezifische Feinheiten beim Verarbeiten von Benutzeraktionen oder dem Verändern des HTML-Dokuments hinter einer einheitlichen Schnittstelle versteckt. Andererseits werden Eigenheiten

[1] https://jquery.com/browser-support/

bezüglich des Designs mit dem GUI-Framework Semantic UI[2] behandelt, wodurch die Oberfläche in den unterstützten Browsern ein einheitliches Design bietet.

Wie auch die Serveranwendung, ist die Weboberfläche in den aktuellen Versionen einiger Browser getestet worden. Hierzu gehören *Mozilla Firefox 40 + 41*, *Google Chrome 44 + 45* und *Microsoft Edge*.

5.1.2. Validierung der generierten Ontologie

Beim Überprüfen der generierten Ontologie kann zwischen einzelnen Teilen weiter differenziert werden. In den nächsten Absätzen wird nach dem Bottom-up-Prinzip von den kleinsten Bestandteilen der Ontologie zur gesamten Ontologie gefolgert.

OWL-Elemente der Features

Wählt der Anwender ein Feature eines bestimmten OWL-Elements aus, soll dieses auch in der Ontologie enthalten sein. Optimal wäre nun, wenn für jedes Feature ein automatisch Test vorhanden wäre, der dessen Funktionalität bestätigt. Diese Tests sind allerdings **nicht** vorhanden.

Beim Generieren werden über die OWL API die OWL-Elemente zur Ontologie hinzugefügt. Der Code hierfür ist durch die OWL-nahen Methoden und Objekte der OWL API sehr kompakt. Im Gegensatz dazu ist der Aufwand zum Lesen der Ontologie mit Hilfe der OWL API unverhältnismäßig groß. Bei über 100 Features wäre dadurch ein zu großer Teil des verfügbaren Zeitrahmens beansprucht worden. Da die einzelnen Features im Programmcode gut voneinander isoliert sind und bei der Entwicklung jedes Feature mindestens einmal in der Ontologie überprüft wurde, wurden automatische Tests hierfür ausgelassen.

OWL-Profile

Die Features stellen die elementaren Bestandteile dar aus denen die Ontologie generiert werden kann und sind nach obiger Annahme korrekt implementiert. Die OWL-Profile, die in dieser Anwendung durch Vorauswahlen von Features realisiert sind, stellen die nächste, zu prüfende Ebene dar.

Die beiden OWL 1-Profile OWL Lite und OWL DL lassen sich manuell sehr einfach prüfen. Für beide Profile wird im *OWL Language Overview* in [MH04] für jedes OWL-Element die eindeutige Aussage getroffen, ob dieses im jeweiligen Profil erlaubt ist. Für diese Profile von OWL 1 habe ich keine automatische Validierung gefunden, weshalb sie nur manuell geprüft wurden.

Für OWL 2-Profile gibt es keine vollständige Auflistung für jedes einzelne OWL-Element. In den *OWL 2 Language Profiles* in [MGH+12] finden sich teils Listen erlaubter Elemente, teils Listen verbotener Elemente und teils Listen von Elementen, für die der Kontext anderer Elemente eine Rolle spielt. Da diese drei Listen zum einen nicht alle möglichen OWL-Elemente abdecken und zum anderen – falls erforderlich – keinen erlaubten Kontext angeben, sind sie für die Validierung von OWL 2-Profilen

[2] https://github.com/Semantic-Org/Semantic-UI#browser-support

nicht ausreichend. Sie werden allerdings durch eine vollständige und komplexere Syntaxbeschreibung im unteren Bereich des Dokuments komplettiert.

Für OWL 2 bietet die OWL API für alle Profile eine automatische Validierung. Beim Implementieren der OWL 2 Profile in der Anwendung verwendete der Einfachheit halber diese Profile anfangs als einzige Prüfung. Als ich die Ergebnisse der automatischen Validierung verifizierte, stieß ich allerdings auf einige Inkonsistenzen zwischen den Prüfungen der OWL API und der Spezifikation in [MGH+12].

Nach mehrfacher Prüfung meiner Ergebnisse begann ich die gesamten Profile allein mit der Spezifikation als Quelle zu implementieren. Anschließend verwendete ich meine Implementierung als Referenz, um die Profile *OWL 2 EL*, *OWL 2 QL* und *OWL 2 RL* der OWL API zu validieren, und meldete die Ergebnisse den Entwicklern auf Github[3]. Die Entwickler prüften meine Funde und bestätigten, dass die OWL API in manchen Fällen fälschlicherweise Warnungen meldete (false positive) und in einem Fall keine Warnung zeigte (false negative). In der für Oktober 2015 angekündigten Version 4.1.0 werden diese Fehler in der OWL API behoben sein.

Während der Diskussion auf Github ergab sich allerdings auch die weitreichendere Erkenntnis, dass die Spezifikation der OWL 2 Profile einen Fehler enthält[4]. Einfach erklärt liegt der Fehler darin, dass ein bestimmter Typ von OWL-Elementen (die Datatypes) nur aus einer in den Profilen beschriebene Menge stammen darf. Gleichzeitig wird aber an einer anderen Stelle erlaubt, dass beliebige Elemente dieses Typs verwendet werden dürfen.

Struktur und Syntax

Bei der Ontologie selbst kann einerseits ihre Struktur und die Kombination der verschiedenen OWL-Elemente betrachtet werden.

Mit der OWL API sollten nur gültige Ontologien erstellt werden können. Das wird unter anderem dadurch erreicht, dass mittels statischer Typisierung alle Restriktionen aus der Spezifikation auf den Java-Code übertragen werden. Es existieren im Quellcode der OWL API aber auch zusätzliche Prüfungen mit denen weitere mögliche Fehlerfälle abgefangen werden.

Die OWL API validiert zwar beim Erstellen und beim Einlesen eine Ontologie; dennoch wäre ein Fehler in beiden Komponenten nicht ausgeschlossen, wenn sie beispielsweise vom selben Entwickler programmiert wurden. Daher wäre eine externe Validierung als zusätzlicher Faktor wünschenswert. Leider existieren hierfür nur der *Manchester OWL Validator*[5] und Protégé [KHM+05], welche allerdings beide die OWL API verwenden. Außer mit der OWL API habe ich die Ontologie in verschiedenen Syntaxen nach bestem Wissen auch selbst auf ihre Korrektheit bezüglich der OWL-Spezifikation überprüft.

Die Syntax der Ontologie lässt sich im Gegensatz zur Struktur leichter überprüfen, da sie nicht unbedingt OWL-spezifisch ist. Für die obligatorische RDF/XML-Syntax konnte der *RDF Validator*[6]

[3] https://github.com/owlcs/OWL-API/issues/435
[4] https://lists.w3.org/Archives/Public/public-owl-comments/2015Aug/0000.html
[5] http://mowl-power.cs.man.ac.uk:8080/validator/
[6] http://www.w3.org/RDF/Validator/

des W3C verwendet werden. Da ich für die anderen Syntaxen keine offiziellen Validatoren finden konnte, sind sie nicht extra überprüft worden.

Zusammengefasst kann man also sagen, dass die Korrektheit zur Spezifikation stark von der OWL API abhängt. Sie wird aber auch durch weitere eigene Prüfungen bestätigt.

5.2. Testen von Ontologievisualisierungen

Mit der graphbasierten Web-Visualisierung *WebVOWL*, der hierarchischen Visualisierung *OWLViz* und dem auf UML basierenden *OWLGrEd Ontology Visualizer* sollen im Folgenden die Tauglichkeit der Software bewertet werden.

Die Entscheidung fiel auf diese drei Visualisierungen, da sie sich stark in ihren Ansätzen und in der Unterstützung von OWL unterscheiden, wodurch eine möglichst vielfältige Testumgebung erreicht wird.

5.2.1. Ablauf

Bei allen Visualisierungen soll nach dem gleichen Ablauf vorgegangen werden, um vergleichbare Ergebnisse zu erhalten. Die Auswahl der Features darf sich bei den Visualisierungen unterscheiden, da auf ihre Fähigkeiten Rücksicht genommen werden soll, denn es macht keinen Sinn, die Visualisierung nicht unterstützter OWL-Elemente zu testen.

1. Die Weboberfläche von OntoBench wird aufgerufen.
2. Es werden Features für die jeweilige Visualisierung gewählt.
3. Die Ontologie wird generiert.
4. Es wird eine von der Visualisierung unterstützte Syntax ausgewählt.
5. Die Ontologie wird über den einfachsten Weg in die Visualisierung geladen (URL oder heruntergeladene Datei).
6. Es wird versucht, mögliche Mängel der Visualisierung zu finden.

5.2.2. WebVOWL

WebVOWL [LLMN14] ist eine webbasierte Visualisierung, die Ontologien als Graphen visualisiert. Ihre visuelle Notation ist in der VOWL-Spezifikation[7] definiert. Ontologien werden von WebVOWL nicht direkt eingelesen und dargestellt, da das Verarbeiten der Ontologie im Browser zu aufwendig ist. Stattdessen wird eine Ontologie im Hintergrund von der serverseitigen Anwendung OWL2VOWL in ein für WebVOWL lesbares Format konvertiert. Getestet wird WebVOWL in der Version *beta 0.4.0*.

[7] http://vowl.visualdataweb.org/v2/

WebVOWL kann nur einen Teil von OWL visualisieren, welcher in der zugehörigen Spezifikation beschrieben ist. Die Syntax einer Ontologie kann aber beliebig gewählt werden, da OWL2VOWL selbst die OWL API verwendet und somit alle Syntaxen lesen kann, die zur Auswahl stehen.

Da ich einer der Hauptentwickler von WebVOWL bin, ist mir die Visualisierung sehr vertraut. Für die Auswertung hat das den Vorteil, dass mir auch kleine Abweichungen von der spezifizierten Funktionalität auffallen.

Ablauf

In der Weboberfläche der Anwendung wähle ich die von VOWL unterstützten Elemente aus. Anschließend generiere ich die Ontologie und kopiere die lange URL. In der Visualisierung wird dann die URL in der Oberfläche eingefügt und die Ontologie in die Visualisierung geladen.

In der visualisierten Ontologie achte ich zunächst auf Auffälligkeiten. Danach wechsele ich in der Anwendung zur Übersicht der Features, um alle einzeln nacheinander überprüfen zu können.

Auffälligkeiten

Beim ersten Blick auf die visualisierte Ontologie fiel mir die Konstellation aus Abb. 5.1 auf, welche nicht auftreten dürfte. Laut der Spezifikation von VOWL müssten `owl:Thing` und `owl:Nothing` gleich dargestellt werden. Die beiden kreisförmigen Knoten unterscheiden sich jedoch deutlich.

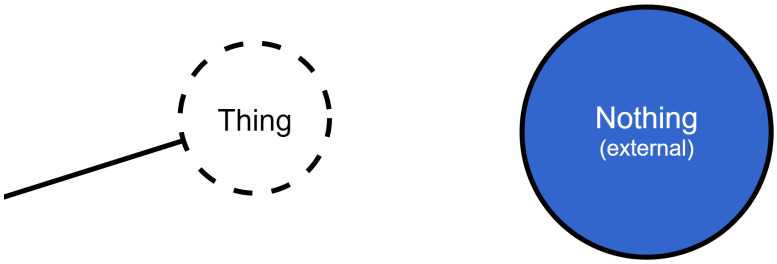

Abbildung 5.1.: Laut der Spezifikation von WebVOWL müssten `Thing` und `Nothing` gleich dargestellt werden.

Ein weiterer Fehler ist, dass keine OWL-Elemente vom Feature `owl:complementOf` angezeigt werden. Außerdem wurde das Feature `owl:maxCardinality` nicht korrekt angezeigt. In Abb. 5.2 sieht man deutlich, dass auf der rechten Seite keine Kardinalitäten angezeigt werden.

5.2.3. OWLViz

OWLViz[8] visualisiert nur Hierarchien von Klassen. Die Visualisierung kann über ein Protégé-Plugin verwendet werden.

[8] http://protegewiki.stanford.edu/wiki/OWLViz

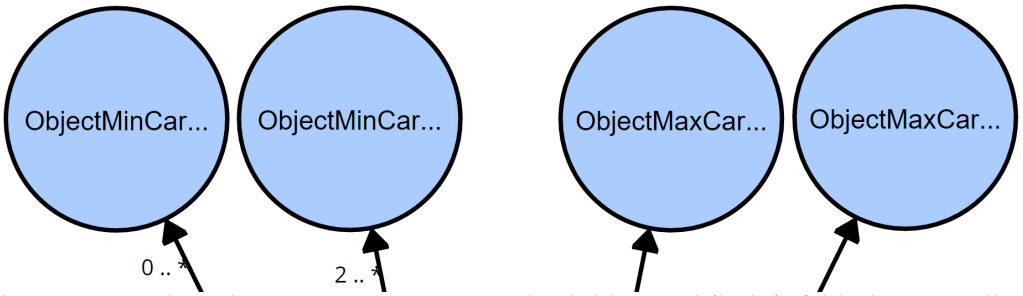

Abbildung 5.2.: Während `owl:minCardinality` abgebildet wird (links), fehlt die Darstellung von `owl:maxCardinality`.

Die generierten Ontologien sind nicht direkt zum Testen dieser Visualisierung geeignet, da sie hauptsächlich aus flachen Konstrukten bestehen. Es kann aber trotzdem getestet werden, ob OWLViz alle möglichen Deklarationen einer Klasse visualisieren kann.

Ablauf

In der Weboberfläche wähle ich mit wenigen Klicks auf *Select All* alle möglichen Features aus, die Klassen beschreiben können. Nach dem Generieren, lade ich die Ontologie herunter und öffne sie direkt in Protégé. OWLViz zeigt dort eine Liste aller visualisierten Klassen, welche ich mit den ausgewählten Features abgleiche.

Auffälligkeiten

Beim Evaluieren der visualisierten Ontologie sind mir keine Fehler aufgefallen. Das liegt unter anderem auch daran, dass die Visualisierung schon seit mindestens 2005 existiert und somit viele Fehler bereits gefunden wurden. Gleichzeitig deckt die Visualisierung auch nicht alle möglichen OWL-Elemente, sondern nur einen kleineren Bereich, ab. Außerdem werden Hierarchien visualisiert, die mit OntoBench nicht gut getestet werden können.

5.2.4. OWLGrEd

Der OWLGrEd Ontology Visualizer [LGB14] ist eine Webanwendung, die Ontologien mit einer UML-basierten Notation visualisiert. Für diese Visualisierung ist eine Notation[9] der unterstützten Elemente vorhanden.

[9] http://owlgred.lumii.lv/notation

Ablauf

In der Weboberfläche wähle ich wie bei OWLViz über die *Select All*-Schaltflächen die Kategorien aus, die von der Visualisierung unterstützt werden. Anschließend generiere ich die Ontologie und lade sie herunter. Auf der Webseite von OWLGrEd lade ich die generierte Ontologie hoch und lasse sie visualisieren. Da mir OWLGrEd unbekannt ist, durchstöbere ich die Visualisierung und versuche anhand der Bezeichner der OWL-Elemente Auffälligkeiten zu finden.

Auffälligkeiten

Beim ersten Fund handelt es sich um eine Inkonsistenz in der Visualisierung. In der Ontologie sind drei Properties enthalten, die alle äquivalent sind. Diese Properties sind in Abb. 5.3 abgebildet. Mir ist aufgefallen, dass nur bei einer Property die anderen beiden äquivalenten Properties angegeben sind (grüner Rahmen). Bei den anderen beiden Properties fehlt die zweite äquivalente Property (roter Rahmen).

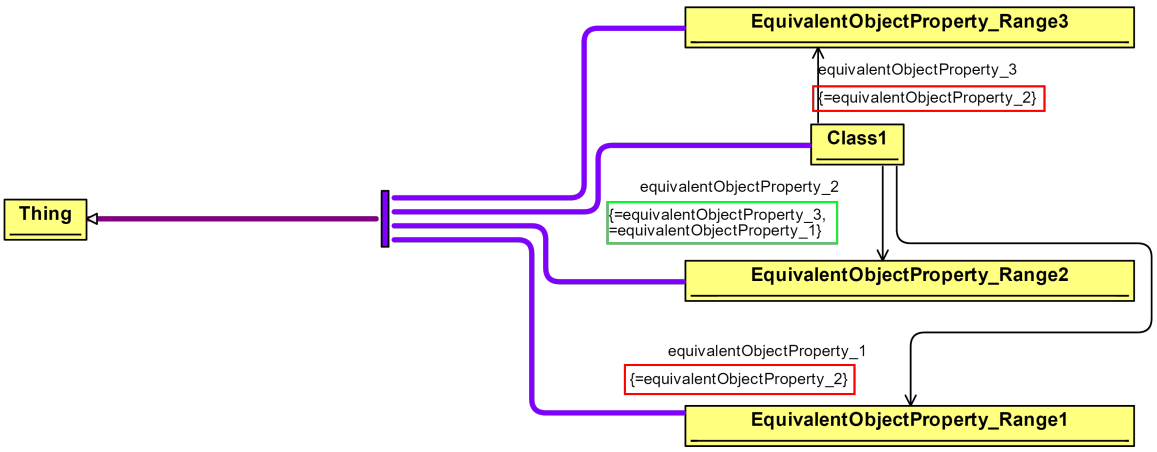

Abbildung 5.3.: Erwartungsgemäß sollten die drei hervorgehobenen Kästchen jeweils die zwei Elemente enthalten.

An einer anderen Stelle handelt es sich um ein ähnliches Problem wie beim ersten Fund. Da es hier allerdings keine Stelle gibt, an der die Informationen korrekt angezeigt werden, stufe ich es als einen Fehler ein. In Abb. 5.4 sollten im rot eingerahmten Bereich eigentlich `hasKeyProperty_1` und `hasKeyProperty_2` stehen. Die Visualisierung scheint allerdings nur einen Wert für den Key anzeigen zu können, obwohl laut offiziellen OWL-Dokumenten mehrere Keys erlaubt sind[10].

[10] http://www.w3.org/TR/owl2-new-features/#F9:_Keys

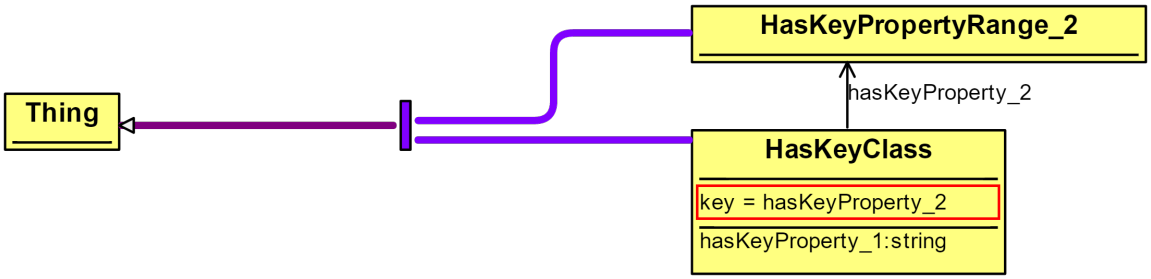

Abbildung 5.4.: Die Visualisierung zeigt nur eine der beiden Properties als Key an.

6. Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde OntoBench, eine Anwendung zum Generieren von Benchmarkontologien für Ontologievisualisierungen, konzipiert und implementiert. Mit diesen Ontologien kann eine Visualisierung primär auf ihre Konformität zur OWL-Spezifikation getestet werden.

In Kapitel 3 wurde zuerst der Umfang der generierten Ontologien definiert und anschließend die Struktur genauer festgelegt. Darauf aufbauend wurde die Entscheidung zur Wahl einer Webanwendung begründet und die Funktionalität dieser spezifiziert. Anhand dieses Konzepts ist OntoBench, wie in Kapitel 4 beschrieben, implementiert worden. Die Anwendung und der gewählte Ansatz wurden abschließend in Kapitel 5 an drei Ontologievisualisierungen getestet und die gesammelten Erfahrungen dokumentiert.

6.1. Fazit

Bei der Evaluierung des Ansatzes anhand der drei Visualisierungen konnten mit verschiedenen generierten Ontologien einige Fehler gefunden werden. Die gute Benennung der OWL-Konstrukte, die zu den einzelnen Features gehören, hat das Überprüfen einzelner Features wesentlich erleichtert. Wenn die Ontologien zu viele Features enthielten, fiel der Abgleich mit der Visualisierung oft schwer, da sie oft zu komplex wurde. Hier konnten als Abhilfe mehrere kleinere Ontologien generiert werden, die eine übersichtlichere Darstellung erzeugten.

Aufgrund der breiten Verwendung der OWL API in allen getesteten Visualisierungen war für keine von diesen eine spezielle Syntax der Ontologie erforderlich. Ebenso hat die Möglichkeit, bestimmte OWL-Profile mit der Ontologie zu testen, keine Anwendung finden können, da die Angaben der OWL-Unterstützung nur auf Ebene der Kategorien getroffen wurden. Hier hat sich jedoch die Auswahlmöglichkeit mehrerer Features einer Kategorie bezahlbar gemacht.

Es fiel unter anderem auf, dass vor allem bei größeren Ontologien eine Checkliste der in der Ontologie enthaltenen Features hilfreich gewesen wäre, mit welcher man Visualisierungen systematischer überprüfen könnte. Dadurch wäre es auch unwahrscheinlicher geworden, dass beim Abgleichen einzelne Features vergessen werden, die zwar in der Ontologie enthalten sind, aber nicht in der Visualisierung angezeigt werden.

Die generierten Ontologien decken einen breiten Rahmen an Testfällen für eine Ontologievisualisierung ab. Dennoch sind zum Verifizieren von möglichen Fehlern gelegentlich speziellere OWL-Konstrukte erforderlich gewesen. Für diesen Fall ist es aber möglich mit Hilfe von einem Ontologie-Editor wie Protégé Detailanpassungen an einer generierten Ontologie im Nachhinein manuell vorzunehmen.

6.2. Ausblick

Beim Testen der Visualisierungen haben sich noch weitere Möglichkeiten gezeigt, mit denen die Anwendung verbessert werden könnte. Diese werden mit weiteren Ideen, mit denen der Ansatz verbessert werden könnte, im Folgenden beschrieben.

6.2.1. Generierte Features auflisten

Damit die Visualisierung mit einer generierten Ontologie verglichen werden kann, muss momentan noch auf die Auswahl aus der Feature-Übersicht zurückgegriffen werden, um einen Überblick über den Inhalt der Ontologie zu erhalten. Wenn eine Liste mit den generierten Features zusammen mit der Ontologie abrufbar wäre, könnte diese auch gespeichert werden, um eine Visualisierung zu einem späteren Zeitpunkt zu testen. Eine weitere Verbesserung wäre, wenn diese Liste auch im Nachhinein durch die URL beziehungsweise anhand der ID abgerufen werden könnte.

6.2.2. OWL-Elemente von Features anzeigen

Beim Abgleich der Visualisierung mit der generierten Ontologie und den zugehörigen Features, müssen momentan die genauen OWL-Elemente erraten werden. Durch eine Benennung der OWL-Elemente wird zwar klar, zu welchem Feature sie gehören; welche Elemente ein Feature umfasst, ist dabei allerdings unbekannt.

Würde ergänzend zum vorigen Verbesserungsvorschlag auch eine Liste der OWL-Elemente eines Features angezeigt werden, könnte die Visualisierung explizit nach diesen durchsucht werden. Damit würde auffallen, wenn eine Visualisierung ein Feature nicht vollständig anzeigt.

6.2.3. Features parametrisieren

An einigen Stellen hat sich gegen Ende der Implementierung und auch beim Evaluieren gezeigt, dass eine Parametrisierung der einzelnen Features eine wichtige Erweiterung darstellt. So könnten beispielsweise Kardinalitäten, Datentypen oder auch die Anzahl an äquivalenten Elementen oder Individuals parametrisiert werden. Dadurch würden separate Features für Kardinalitäten der unterschiedlichen OWL-Profile wegfallen. Die Parameter werden aber auch für flexiblere Annotationen oder Properties einer Ontologie wie beispielsweise `owl:imports` benötigt.

Die große Herausforderung hierbei stellt vor allem die Validierung der vielen möglichen Parametertypen und der Umgang mit eventuellen Fehlern dar. Ansonsten könnte diese Anpassung ohne große strukturelle Änderungen vorgenommen werden.

6.2.4. Mehr visualisierungsspezifische Testfälle

Bisher stehen neben den Features für OWL-Konstrukte auch wenige visualisierungspezifischere Features zur Auswahl. In einer separaten Arbeit könnten bisherige Visualisierungsansätze für Ontologien betrachtet werden und aus diesen neue Testfälle im Zusammenhang mit bestimmten OWL-Konstrukten abgeleitet werden. Eventuell könnten auch Entwickler von Ontologievisualisierungen befragt werden, ob ihnen bei der Entwicklung mögliche Fallstricke aufgefallen oder begegnet sind.

A. Unterstützte OWL-Elemente

OWL-Konstrukt	Verfügbar
Anonymous Individual	✔
owl:AllDifferent	✔
owl:AllDisjointClasses	✔
owl:AllDisjointProperties	✔
owl:allValuesFrom	✔
owl:annotatedProperty	✘
owl:annotatedSource	✘
owl:annotatedTarget	✘
owl:Annotation	✘
owl:AnnotationProperty	✔
owl:assertionProperty	✔
owl:AsymmetricProperty	✔
owl:Axiom	✘
owl:backwardCompatibleWith	✘
owl:bottomDataProperty	✔
owl:bottomObjectProperty	✔
owl:cardinality	✔
owl:Class	✔
owl:complementOf	✔
owl:dataComplementOf	✔
owl:DatatypeProperty	✔
owl:deprecated	✔
owl:differentFrom	✘
owl:disjointUnionOf	✔
owl:disjointWith	✔
owl:equivalentClass	✔
owl:equivalentProperty	✔
owl:FunctionalProperty	✔
owl:hasKey	✔
owl:hasSelf	✔
owl:hasValue	✔
owl:imports	✘
owl:incompatibleWith	✘
owl:intersectionOf	✔
owl:inverseOf	✔
owl:IrreflexiveProperty	✔
owl:InverseFunctionalProperty	✔
owl:maxCardinality	✔
owl:maxQualifiedCardinality	✔
owl:members	✔
owl:minCardinality	✔

OWL-Konstrukt	Verfügbar
owl:minQualifiedCardinality	✔
owl:NamedIndividual	✔
owl:NegativePropertyAssertion	✔
owl:Nothing	✔
owl:ObjectProperty	✔
owl:onClass	✔
owl:onDataRange	✔
owl:onDatatype	✔
owl:oneOf	✔
owl:onProperties	✘
owl:onProperty	✔
owl:Ontology	✔
owl:priorVersion	✘
owl:propertyChainAxiom	✔
owl:propertyDisjointWith	✔
owl:qualifiedCardinality	✔
owl:ReflexiveProperty	✔
owl:Restriction	✔
owl:sameAs	✔
owl:someValuesFrom	✔
owl:sourceIndividual	✔
owl:targetIndividual	✔
owl:targetValue	✔
owl:SymmetricProperty	✔
owl:Thing	✔
owl:TransitiveProperty	✔
owl:topDataProperty	✔
owl:topObjectProperty	✔
owl:unionOf	✔
owl:versionInfo	✔
owl:versionIRI	✘
owl:withRestrictions	✔
rdfs:comment	✔
rdfs:Datatype	✔
rdfs:domain	✔
rdfs:isDefinedBy	✘
rdfs:label	✔
rdfs:range	✔
rdfs:seeAlso	✘
rdfs:subClassOf	✔
rdfs:subPropertyOf	✔

Literaturverzeichnis

[Ber15] D. Berger. Chrome-Browser pausiert Flash-Inhalte. *heise online*, 1 September 2015. URL http://www.heise.de/newsticker/meldung/Chrome-Browser-pausiert-Flash-Inhalte-2796860.html. (Zitiert auf Seite 32)

[BHH+04] S. Bechhofer, F. van Harmelen, J. H. I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein. *OWL Web Ontology Language Reference*. W3C Recommendation, 10 February 2004. URL http://www.w3.org/TR/2004/REC-owl-ref-20040210/. (Zitiert auf Seite 12)

[BKMPS12] J. Bao, E. F. Kendall, D. L. McGuinness, P. F. Patel-Schneider, Herausgeber. *OWL 2 Web Ontology Language: Quick Reference Guide (Second Edition)*. W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-quick-reference-20121211/. (Zitiert auf Seite 21)

[BLHL+01] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001. (Zitiert auf Seite 11)

[CR04] J. J. Carroll, J. D. Roo, Herausgeber. *OWL 2 Web Ontology Language Conformance (Second Edition)*. W3C Recommendation, 10 February 2004. URL http://www.w3.org/TR/2004/REC-owl-test-20040210/. (Zitiert auf Seite 14)

[DCM12] DCMI Usage Board. *DCMI Metadata Terms*. DCMI Recommendation, 14 June 2012. URL http://dublincore.org/documents/2012/06/14/dcmi-terms/. (Zitiert auf den Seiten 12 und 22)

[GPH05] Y. Guo, Z. Pan, J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005. URL http://dx.doi.org/10.1016/j.websem.2005.06.005. (Zitiert auf Seite 13)

[HB11] M. Horridge, S. Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011. URL http://dx.doi.org/10.3233/SW-2011-0025. (Zitiert auf Seite 37)

[HKP+12] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, S. Rudolph, Herausgeber. *OWL 2 Web Ontology Language: Primer (Second Edition)*. W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-primer-20121211/. (Zitiert auf Seite 12)

[HLNE14a] F. Haag, S. Lohmann, S. Negru, T. Ertl. OntoViBe 2: Advancing the Ontology Visualization Benchmark. In P. Lambrix, E. Hyvönen, E. Blomqvist, V. Presutti, G. Qi, U. Sattler, Y. Ding, C. Ghidini, Herausgeber, *Knowledge Engineering and Knowledge Management - EKAW*

2014 Satellite Events, VISUAL, EKM1, and ARCOE-Logic, Linköping, Sweden, November 24-28, 2014. Revised Selected Papers., Band 8982 von *Lecture Notes in Computer Science*, S. 83–98. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-17966-7_9. (Zitiert auf den Seiten 12 und 20)

[HLNE14b] F. Haag, S. Lohmann, S. Negru, T. Ertl. OntoViBe: An Ontology Visualization Benchmark. In V. Ivanova, T. Kauppinen, S. Lohmann, S. Mazumdar, C. Pesquita, K. Xu, Herausgeber, *Proceedings of the International Workshop on Visualizations and User Interfaces for Knowledge Engineering and Linked Data Analytics co-located with 19th International Conference on Knowledge Engineering and Knowledge Management, VISUAL@EKAW 2014, Linköping, Sweden, November 24, 2014.*, Band 1299 von *CEUR Workshop Proceedings*, S. 14–27. CEUR-WS.org, 2014. URL http://ceur-ws.org/Vol-1299/paper2.pdf. (Zitiert auf Seite 12)

[KHM+05] H. Knublauch, M. Horridge, M. A. Musen, A. L. Rector, R. Stevens, N. Drummond, P. W. Lord, N. F. Noy, J. Seidenberg, H. Wang. The Protege OWL Experience. In B. C. Grau, I. Horrocks, B. Parsia, P. F. Patel-Schneider, Herausgeber, *Proceedings of the OWLED*05 Workshop on OWL: Experiences and Directions, Galway, Ireland, November 11-12, 2005*, Band 188 von *CEUR Workshop Proceedings*. CEUR-WS.org, 2005. URL http://ceur-ws.org/Vol-188/sub14.pdf. (Zitiert auf den Seiten 14 und 57)

[LGB14] R. Liepins, M. Grasmanis, U. Bojars. OWLGrEd Ontology Visualizer. In R. Verborgh, E. Mannens, Herausgeber, *Proceedings of the ISWC Developers Workshop 2014, co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 19, 2014.*, Band 1268 von *CEUR Workshop Proceedings*, S. 37–42. CEUR-WS.org, 2014. URL http://ceur-ws.org/Vol-1268/paper7.pdf. (Zitiert auf Seite 60)

[LLMN14] S. Lohmann, V. Link, E. Marbach, S. Negru. WebVOWL: Web-based Visualization of Ontologies. In P. Lambrix, E. Hyvönen, E. Blomqvist, V. Presutti, G. Qi, U. Sattler, Y. Ding, C. Ghidini, Herausgeber, *Knowledge Engineering and Knowledge Management - EKAW 2014 Satellite Events, VISUAL, EKM1, and ARCOE-Logic, Linköping, Sweden, November 24-28, 2014. Revised Selected Papers.*, Band 8982 von *Lecture Notes in Computer Science*, S. 154–158. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-17966-7_21. (Zitiert auf den Seiten 37 und 58)

[LNHE14] S. Lohmann, S. Negru, F. Haag, T. Ertl. VOWL 2: User-Oriented Visualization of Ontologies. In K. Janowicz, S. Schlobach, P. Lambrix, E. Hyvönen, Herausgeber, *Knowledge Engineering and Knowledge Management - 19th International Conference, EKAW 2014, Linköping, Sweden, November 24-28, 2014. Proceedings*, Band 8876 von *Lecture Notes in Computer Science*, S. 266–281. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-13704-9_21. (Zitiert auf Seite 18)

[MGH+12] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, Herausgeber. *OWL 2 Web Ontology Language Profiles (Second Edition).* W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/. (Zitiert auf den Seiten 56 und 57)

[MH04] D. L. McGuinness, F. van Harmelen, Herausgeber. *OWL Web Ontology Language Overview*. W3C Recommendation, 10 February 2004. URL http://www.w3.org/TR/2004/REC-owl-features-20040210/#s2. (Zitiert auf Seite 56)

[MPSP12] B. Motik, P. F. Patel-Schneider, B. Parsia, Herausgeber. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition)*. W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/. (Zitiert auf Seite 21)

[MYQ+06] L. Ma, Y. Yang, Z. Qiu, G. T. Xie, Y. Pan, S. Liu. Towards a Complete OWL Ontology Benchmark. In Y. Sure, J. Domingue, Herausgeber, *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*, Band 4011 von *Lecture Notes in Computer Science*, S. 125–139. Springer, 2006. URL http://dx.doi.org/10.1007/11762256_12. (Zitiert auf Seite 14)

[SHKG12] M. Smith, I. Horrocks, M. Krötzsch, B. Glimm, Herausgeber. *OWL 2 Web Ontology Language Conformance (Second Edition)*. W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-conformance-20121211/. (Zitiert auf Seite 14)

[W3C12] W3C OWL Working Group, Herausgeber. *OWL 2 Web Ontology Language: Document Overview (Second Edition)*. W3C Recommendation, 11 December 2012. URL http://www.w3.org/TR/2012/REC-owl2-overview-20121211/. (Zitiert auf den Seiten 12 und 23)

[WGQH05] S. Wang, Y. Guo, A. Qasem, J. Heflin. Rapid Benchmarking for Semantic Web Knowledge Base Systems. In Y. Gil, E. Motta, V. R. Benjamins, M. A. Musen, Herausgeber, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, Band 3729 von *Lecture Notes in Computer Science*, S. 758–772. Springer, 2005. URL http://dx.doi.org/10.1007/11574620_54. (Zitiert auf Seite 13)

Alle URLs wurden zuletzt am 10. 10. 2015 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