WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

Bachelorarbeit

Deutscher Titel der Bachelorarbeit	Ein GWAP Plugin zur Validierung von Ontologien für den Protégé Ontology Editor
Englischer Titel der Bachelorarbeit	A GWAP plugin for the validation of ontologies for the Protégé Ontology Editor
Verfasser/in **Familienname, Vorname(n)**	Hanika, Florian
Matrikelnummer	h0951511
Studium	Bachelorstudium Wirtschafts- und Sozialwissenschaften
Beurteiler/in **Titel, Vorname(n), Familienname**	Mag. Dr. Gerhard Wohlgenannt

Hiermit versichere ich, dass

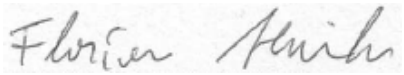
1. ich die vorliegende Bachelorarbeit selbständig und ohne Verwendung unerlaubter Hilfsmittel verfasst habe. Alle Inhalte, die direkt oder indirekt aus fremden Quellen entnommen sind, sind durch entsprechende Quellenangaben gekennzeichnet.

2. die vorliegende Arbeit bisher weder im In- noch im Ausland zur Beurteilung vorgelegt bzw. veröffentlicht worden ist.

3. diese Arbeit mit der beurteilten bzw. in elektronischer Form eingereichten Bachelorarbeit übereinstimmt.

4. (nur bei Gruppenarbeiten): die vorliegende Arbeit gemeinsam mit Vorname(n), Familienname(n) entstanden ist. Die Teilleistungen der einzelnen Personen sind kenntlich gemacht, ebenso wie jene Passagen, die gemeinsam erarbeitet wurden.

Datum 05.10.2013



Unterschrift

Institut für Informationswirtschaft

Wirtschaftsuniversität Wien

Bachelorarbeit

Ein GWAP Plugin zur Validierung von Ontologien für den Protégé Ontology Editor

Florian Hanika

Matrikelnummer: h0951511

Bachelorstudium Wirtschafts- und Sozialwissenschaften

Studienzweig Wirtschaftsinformatik

Betreuer: Mag. Dr. Gerhard Wohlgenannt

Kurzfassung

Diese Arbeit beschäftigt sich mit der Entwicklung eines Plugins für den Protégé Ontology Editor zur Validierung von Ontologien mit Hilfe eines Games with a purpose (GWAP). Dies hat das Ziel, die Automatisierung und damit auch Skalierbarkeit im Semantic Web, einer Weiterentwicklung des heutigen World Wide Web, voranzutreiben. Unter einer Ontologie versteht man eine nach bestimmten Regeln strukturierte Darstellung komplexer Wissensbeziehungen. Protégé ist der bekannteste und verbreitetste Editor zur Darstellung und Bearbeitung solcher Ontologien. Ein GWAP ist ein oft mit Web 2.0 Technologien verbundenes Spiel, in dem die Benutzer spielerisch dabei helfen, ein komplexes Problem zu lösen, welches für die Aufbereitung im Spiel in viele kleine Tasks zerlegt wurde.

Zu Beginn dieser Arbeit werden alle relevanten grundlegenden Konzepte und Technologien näher erläutert. Dazu zählen das Semantic Web, Ontologien (inklusive der Ontologiesprache OWL und den Möglichkeiten des automatisierten Ontology Learning), der Protégé Ontology Editor, GWAPs, die Programmiersprache Java, auf der Protégé und die entsprechende Plugin Entwicklung komplett basieren und JSON, ein schlankes Datenaustauschformat, welches in der Kommunikation zwischen Plugin und GWAP verwendet wird.

Danach folgt der praktische Teil der Arbeit, in dem die konkrete Entwicklung des Plugin beschrieben wird. Dafür wird zunächst im Rahmen der Pilotstudie erklärt, wie ein Plugin für Protégé grundsätzlich programmiert werden kann. Von der Einrichtung der Entwicklungsumgebung, über den mindestens notwendigen Aufbau eines Plugins bis hin zur Beschreibung spezieller Eigenheiten bei der Entwicklung der grafischen Benutzeroberfläche wird alles erläutert, was im Allgemeinen für die Entwicklung eines Protégé Plugins benötigt wird. Danach wird detaillierter auf das GWAP Plugin eingegangen. Nach einer Beschreibung des Funktionsumfangs (dazu zählen die Validierung von einzelnen Konzepten, Individuen und Teilbäumen von Konzepten) folgt der Aufbau des Plugins anhand eines Klassendiagramms. Schlussendlich findet noch eine Betrachtung und Analyse spezieller Programmkomponenten (wie zum Beispiel die Manipulation von OWL Komponenten oder die Kommunikation mit dem GWAP über JSON) statt, die jeweils eine eigene Einarbeitung benötigt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Der Entwicklung des sogenannten semantischen Web als Ergänzung und Weiterentwicklung des World Wide Web kommt in der Forschung im Bereich der Webtechnologien seit einem guten Jahrzehnt besondere Bedeutung zu. In diesem Rahmen spielen Ontologien zur Darstellung komplexer Wissensbeziehungen eine wesentliche Rolle. Ein kritischer Faktor für den Erfolg des semantischen Webs wird seine Skalierbarkeit sein, was unter anderem zur Fragestellung führt, wie Ontologien möglichst ressourcenschonend erstellt und validiert werden können.

1.1 Problemstellung und Zielsetzung

Ontologien können sowohl manuell von Experten als auch automatisiert von speziellen Algorithmen, welche das Web, Datenbanken oder andere Datenstrukturen durchsuchen, erstellt werden. Aus wirtschaftlichen und auch den bereits erwähnten Gründen der Skalierbarkeit ist das automatisierte Ontology Learning zu bevorzugen. Jedoch entsteht dadurch das Problem, dass von Maschinen erstellte Ontologien erst von Menschen auf ihre Gültigkeit und Sinnhaftigkeit überprüft werden müssen. Eine Variante, um dieses Problem zu lösen, ist wiederum das Hinzuziehen von Experten, welche die automatisiert erstellten Ontologien überprüfen. Dies ist zwar wesentlich ressourcenschonender als die komplett manuelle Erstellung, hat aber im Kern dasselbe Problem. Ein vielversprechender Lösungsansatz ist hierbei nun das sogenannte Crowd-Sourcing, bei dem diese Aufgabe an eine Vielzahl freiwilliger Benutzer über das Internet ausgelagert werden kann. Eine konkrete Ausprägung dieses Prinzips sind Games with a purpose (GWAPs). Dies sind mit Web 2.0 Technologien beziehungsweise sozialen Plattformen verknüpfte Spiele, in denen die Benutzer bestimmte Aufgaben lösen sollen und dafür zum Beispiel Punkte erhalten.

Diese Arbeit soll dieses Konzept aufgreifen, indem als Zielsetzung die Entwicklung eines Plugins für den bekanntesten und am weitesten verbreiteten Ontologie Editor namens Protégé zur Validierung von Ontologien definiert wird. Das Plugin soll einem Benutzer ermöglichen, einen Teil einer vorhandenen Ontologie gegen eine bestimmte Domäne (wie zum Beispiel Sport, Verreisen oder Essen und Trinken) zu validieren. Zu diesem Zweck markiert der Benutzer ei-

nen Teil der Ontologie, gibt die Domäne und gegebenenfalls zusätzliche Information als Text an, und startet die Validierung. Im Hintergrund werden diese Daten dann an das GWAP gesendet. Sobald ein Ergebnis vorliegt, wird dieses vom GWAP abgerufen und dem Benutzer dargestellt. Fällt die Gültigkeitsprüfung positiv aus, verbleibt die überprüfte Einheit in der Ontologie, fällt sie negativ aus, erhält der Benutzer die Möglichkeit die Einheit zu löschen.

1.1.1 Forschungsleitende Fragestellung

Diese Arbeit soll durch die gewonnenen Erkenntnisse bei der Beantwortung folgender übergeordneter allgemeiner Forschungsfragen behilflich sein:

A1: „Sind Crowd-Sourcing Ansätze dafür geeignet, bei der Erstellung und Validierung von Ontologien hilfreich zu sein?“

A2: „Kann man teure Experten für die Erstellung und Validierung von Ontologien durch automatisierte Ansätze ersetzen, welche durch Crowd-Sourcing unterstützt werden, um eine bessere Skalierbarkeit zu erreichen?“

Da die gänzliche Beantwortung dieser allgemeinen Fragen den Rahmen dieser Arbeit übersteigt, lautet die spezielle Forschungsfrage in diesem Fall:

S1: „Wie kann ein Plugin für den Protégé Ontology Editor zur Validierung von Ontologien mit Hilfe von Games with a purpose entwickelt werden?“

1.2 Motivation

Die Entwicklung des Semantic Web ist nach Meinung des Autors ein interessanter und zukunftsträchtiger Forschungsbereich. Auch wenn es sich bis heute noch nicht flächendeckend durchsetzen konnte, liegt die Vermutung nahe, dass es mittel- bis langfristig eine große Rolle spielen wird. Die vorliegende Arbeit soll mit der Entwicklung des Plugins einen Beitrag leisten, den Erstellungs- und Validierungsprozess von Ontologien, welche ein zentraler Bestandteil des Semantic Web sind, effizienter und einfacher zu gestalten.

1.3 Relevanz

Es wurde bereits erwähnt, dass zu den wichtigsten Faktoren für den langfristigen Erfolg des Semantic Web die Einfachheit und Skalierbarkeit zählen. Das

führte auch beim ursprünglichen World Wide Web zur raschen und wirksamen Verbreitung. Für die einfache Handhabung des Semantic Web benötigt es eine solide Grundlage in Form von klaren Standards, Protokollen und Prozessen. Manche Funktionen müssen für den Endbenutzer intransparent ablaufen. Zum Beispiel werden sich viele Benutzer mangels Zeit und technischer Vorkenntnisse nicht mit der Erstellung, Validierung und Verbesserung von Ontologien auseinander setzen wollen. Jedoch sollte jeder die Vorteile einer solch umfangreichen und verknüpften Wissensbasis nutzen können. Dieses Szenario erreicht man nur mit einem möglichst hohen Grad an Automatisierung. Der Erstellungs- und vor allem Validierungsprozess von Ontologien wird (ohne die Entwicklung einer weit fortgeschrittenen künstlichen Intelligenz) vermutlich niemals vollkommen automatisiert werden können. Jedoch soll das beschriebene Plugin dazu beitragen, diesen Prozess weiter zu verbessern.

1.4 Forschungsüberblick

Zwei vergleichbare Projekte, die sich ebenfalls mit einer Verbesserung des Erstellungs- und Validierungsprozesses von Ontologien beschäftigen, sind Crowdmap und das OntoGame.

Crowdmap versucht, jeweils zwei vorhandene Ontologien mittels menschlicher Unterstützung zu vergleichen, um Übereinstimmungen feststellen zu können. Dafür wird die gesamte Arbeit in kleine Microtasks aufgeteilt, welche dann in CrowdFlower, einer Online-Plattform, die einen Arbeitsmarkt im Sinne des Crowd-Sourcings darstellt, ausgeschrieben werden. Mit den Ergebnissen aus der Crowd können dann die Übereinstimmungen zwischen den beiden Ontologien festgestellt werden [vgl. Sara12, 525 ff.].

OntoGame versucht, den Erstellungs- und Validierungsprozess von Ontologien zusammenzufassen. Dafür wird eine Online-Plattform bereitgestellt, bei der die notwendigen Schritte für die Entwicklung einer Ontologie ebenfalls in Microtasks heruntergebrochen werden, die direkt von der Crowd durchgeführt werden können. Als Anwendungsbeispiele werden etwa das Annotieren, also das Versehen einer Sache mit Anmerkungen, von YouTube-Videos oder eBay-Auktionen, sowie das Verwandeln des Inhalts von Wikipedia in eine große Ontologie angeführt [vgl. SiHe08, 751-761].

2. Grundlegende Konzepte und Literaturüberblick

2.1 Semantic Web

Die Idee und Vision des Semantic Web oder auch semantischen Web wurde von Sir Tim Berners-Lee, dem Direktor des World Wide Web Consortiums (W3C), in einem Artikel vorgestellt, der im Scientific American veröffentlicht wurde. In diesem beschreibt er es mit den Worten: „A new form of Web content that is meaningful to computers“ [siehe Bern01, 29].

Das Problem des heutigen herkömmlichen World Wide Web ist, dass die unüberschaubare Menge an Informationen hauptsächlich für den Menschen gedacht ist, und nicht für die automatisierte und logische Interpretation und Schlussfolgerung durch Maschinen. Durch die Vielzahl an heterogenen Informationen fällt es oft schwer, prinzipiell verfügbare Informationen zu einem Themengebiet zu finden und einheitlich aufzubereiten. Mit dem Semantic Web wird versucht, dieses Problem zu adressieren, indem der aussagekräftige Inhalt, der ansonsten quer über viele Webseiten verteilt ist, in eine Struktur gebracht wird. Dadurch wird versucht eine möglichst hohe Informationsintegration zu erreichen. Ein weiterer Vorteil ist, dass aus einer solchen Struktur leicht anhand von formalen Regeln neues implizites Wissen abgeleitet werden kann. Das Semantic Web dient hierbei nicht als Ersatz sondern vielmehr als Erweiterung zum aktuellen Web, mit der Zielstellung die Zusammenarbeit von Mensch und Maschine zu verbessern. Das Semantic Web sollte nicht mit künstlicher Intelligenz verwechselt werden. Es geht hierbei nicht darum, dass Computer den Inhalt und die Bedeutung von Informationen wie ein Mensch verstehen. Vielmehr geht es um eine strukturierte semantische Repräsentation von Wissen, durch die Maschinen auf Basis von einheitlichen Regeln logische Schlussfolgerungen treffen können. Damit sollen skalierbare Lösungen gefunden werden, die dem Menschen Arbeit abnehmen [vgl. Bern01, 30; Hitz08, 10 ff.].

Für den Erfolg und die Verbreitung des Semantic Web sind einige Voraussetzungen notwendig. Ein möglichst hoher Grad an Standardisierung ist wichtig, dies müssen jedoch offene Standards sein. Diese müssen flexibel und erweiterbar sein. Das W3C hat hier bereits viele Standards zu semantischen Technologien geschaffen, die im nächsten Abschnitt 2.1.1 besprochen werden.

Das Semantic Web ist wie das herkömmliche Web dezentral, um ein unbeschränktes Wachstum zu ermöglichen. Der Preis für ein solch komplexes System ist die Gefahr des Auftretens von Paradoxa und per Definition logisch unbeantwortbaren Fragen. Dies muss jedoch im Sinne der vielseitigen Anwendungsmöglichkeiten in Kauf genommen werden [vgl. Bern01, 30; Hitz08, 11].

2.1.1 Aufbau und Bestandteile des Semantic Web

Der Aufbau des Semantic Web Stacks wird in Abbildung 1 dargestellt. Im Folgenden werden die für diese Arbeit relevanten Konzepte näher beschrieben.

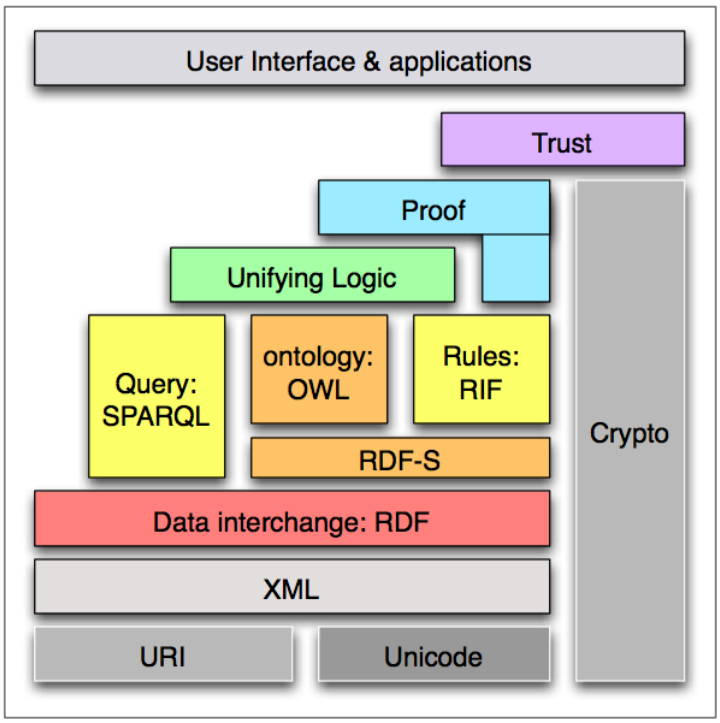


Abbildung 1: Semantic Web Stack [siehe http://www.w3.org/2006/Talks/0811-sb-W3Cemergingtech]

2.1.1.1 URIs und XML

XML ist eine Technologie, die bereits im herkömmlichen Web umfangreiche Verwendung findet. Sie dient zur strukturierten Darstellung von Daten, diese werden dabei mit sogenannten Tags versehen beziehungsweise ausgezeichnet. Zur einfachen Identifikation von Ressourcen werden URIs (Uniform Resource Identifier) verwendet, die weltweit eindeutig sind. Sie bestehen aus einem Schemateil, der den Typ der URI angibt (zum Beispiel http bei einer URL) und einem nachfolgenden Teil, der die Ressource (oft in hierarchischem Aufbau) näher beschreibt, getrennt durch einen Doppelpunkt. XML selbst besitzt keine Semantik, zum Beispiel besteht keine für Maschinen verständliche Bezie-

hung zwischen einzelnen Elementen. Mit XML wird nur die Struktur der Daten vorgegeben, dies dient als Basis für die darauf aufsetzenden semantischen Konzepte [vgl. Bern01, 32; Hitz08, 25-30].

2.1.1.2 RDF und RDF-S

Das Resource Description Framework (RDF) dient nun dazu, die in XML strukturierten Daten näher zu beschreiben und ihnen dadurch eine Bedeutung zu geben. Das Ziel ist ein Austausch von Daten durch Anwendungen im Web, ohne dass die Bedeutung dieser verloren geht. Zu diesem Zweck werden mit RDF Aussagen formuliert, die in Form sogenannter Tripels serialisiert werden. Letztere bestehen dabei immer aus Subjekt, Prädikat und Objekt. Ein Beispiel für eine solche Aussage ist: Florian Hanika (Subjekt) ist der Autor von (Prädikat) dieser Bachelorarbeit (Objekt). Dies ist ein guter Weg, Daten zu beschreiben, die durch Maschinen verarbeitet werden sollen. Betrachtet man alle verknüpften Tripel in einer Struktur, ergibt sich ein System von gerichteten Graphen. Letztere eignen sich für die Darstellung dezentraler Informationen weitaus besser als Bäume, die zu strikt dafür sind [vgl. Bern01, 32; Hitz08, 35-40].

Durch die Verwendung von Vokabularen wie zum Beispiel FOAF (Friend of a Friend, wird für die Beschreibung von Personen benutzt), können thematisch zusammengehörende Ressourcen (durch ihre URIs) zusammengefasst werden. Diese Vereinheitlichung erhöht den Wiederverwendungswert und verhindert, dass Arbeit doppelt gemacht wird [vgl. Hitz08, 47 ff.].

RDF-S ist eine Schema-Sprache, die auf RDF aufsetzt. Mit ihr ist es möglich, terminologisches Wissen abzubilden und einfache Ontologien zu erstellen. Zu diesem Zweck werden zum Beispiel Klassen und Instanzen eingeführt, mit denen Ressourcen typisiert werden können. Zum Beispiel kann Florian Hanika (Subjekt) vom Typ (Prädikat) Student (Objekt, in diesem Fall die Klasse) sein [vgl. Hitz08, 66 ff.].

2.1.1.3 OWL

Die Web Ontology Language (OWL) soll hier nur kurz erwähnt werden, da sie im nächsten Kapitel 2.2 gemeinsam mit Ontologien im Allgemeinen ausführlich behandelt wird. In diesem Kontext soll nur der Zusammenhang mit den restlichen Technologien des Semantic Web Stacks erläutert werden. Wie bereits

erwähnt können mit RDF-S einfache Ontologien (sogenannte lightweight Ontologien) erstellt werden. Um komplexere Zusammenhänge darstellen zu können, benötigt es jedoch eine ausdrucksstärkere Sprache wie OWL. OWL setzt auf RDF-S auf und basiert auf der Prädikatenlogik erster Stufe. Mit ihr kann durch logisches Schlussfolgern aus bestehendem Wissen neues implizites Wissen geschaffen werden [vgl. Hitz08, 125].

2.1.2 Verbreitung des Semantic Web

Ähnlich wie bei der Entwicklung von HTTP am CERN begann die Anwendung des Semantic Web auch in kleinen spezialisierten Bereichen wie den E-Science Communities. Da die Verbreitung schwierig ist, sind solche kleinen Communities als early adopters sehr wichtig. Heute werden das Semantic Web und Ontologien bereits in einigen Branchen verwendet, dazu zählen vor allem die Medizin und Genetik. Aber auch in anderen naturwissenschaftlichen Bereichen wie Gewässerkunde, Klimakunde, Umweltschutz oder der Meeresforschung besteht ein wachsender Bedarf dafür [vgl. Shad06, 96-99].

2.2 Ontologien, OWL und Ontology Learning

Der Begriff Ontologie kommt ursprünglich aus der Philosophie und entspricht einer systematischen Darstellung und Beschreibung allen Seins, also jeglicher Existenz. In der Informatik dient eine Ontologie dazu, Wissen durch IT-Systeme unterstützt auszutauschen. Dafür bedarf es einer klaren Struktur und klaren Konzepten, da in einer von heterogenen IT-Systemen geprägten Welt die Interoperabilität ein enorm wichtiger Faktor ist. Dabei muss die technische Aufbereitung von Wissen unabhängig vom Inhalt sein, das heißt unterschiedliche Themenbereiche werden anhand derselben Regeln und mit denselben Methoden dargestellt. Zusammenfassend kann man sagen, dass Ontologien zur Darstellung komplexer Wissensbeziehungen verwendet werden und sie als „explizite Spezifizierung einer Konzeptualisierung“ betrachten. Zu den Designkriterien von Ontologien zählt deren Klarheit, Kohärenz, Skalierbarkeit, minimaler Encoding Bias und minimales Ontological Commitment. Zwischen diesen Kriterien besteht ein Tradeoff, das heißt für die Verbesserung eines Kriteriums muss eventuell eine Verschlechterung eines anderen Kriteriums in Kauf genommen werden [vgl. Grub95, 907-910].

Ontologien sind ein zentraler Bestandteil des Semantic Web. Nur durch ihre strukturierte Darstellung von Wissensbeziehungen wird es einerseits möglich, Wissen effizient elektronisch auszutauschen und andererseits aus vorhandenem Wissen neues implizites Wissen automatisiert abzuleiten. Wie im vorigen Kapitel unter 2.1.1.3 erwähnt, werden Ontologien im Semantic Web in OWL beschrieben. Im folgenden Abschnitt 2.2.1 wird an Beispielen der Travel-Ontologie [siehe Trav13], die Wissen im Bereich des Verreisens zusammenfasst, der Aufbau der Sprache erklärt.

2.2.1 OWL Aufbau

Am Beginn steht immer der Kopf einer Ontologie. Nach der Angabe aller Namensräume können allgemeine Informationen über die Ontologie wie zum Beispiel Versionsinformationen und Kommentare angegeben werden. Letzteres wird in Abbildung 2 dargestellt [vgl. Hitz08, 128 f.].

```
<owl:Ontology rdf:about="http://www.owl-ontologies.com/travel.owl">
        <owl:versionInfo rdf:datatype="&xsd;string">
                1.0 by Holger Knublauch (holger@smi.stanford.edu)
        </owl:versionInfo>
        <rdfs:comment rdf:datatype="&xsd;string">
                An example ontology for tutorial purposes.
        </rdfs:comment>
</owl:Ontology>
```

Abbildung 2: OWL Ontology Head [siehe Trav13]

Danach folgen die Object und Data Properties, diese entsprechen in RDF Tripeln dem Prädikat. Während Object Properties zwei Individuen miteinander verbinden, verbindet ein Data Property ein Individuum mit einem Datenwert (zum Beispiel eine Person mit einer E-Mail Adresse). In Abbildung 3 wird ein Object Property dargestellt, das einen Urlaubszielort mit einer Unterkunft verknüpft [vgl. Hitz08, 130].

```
<owl:ObjectProperty rdf:about=
        "http://www.owl-ontologies.com/travel.owl#hasAccommodation">
        <rdfs:range rdf:resource=
                "http://www.owl-ontologies.com/travel.owl#Accommodation"/>
        <rdfs:domain rdf:resource=
                "http://www.owl-ontologies.com/travel.owl#Destination"/>
</owl:ObjectProperty>
```

Abbildung 3: OWL Object Property [siehe Trav13]

Nun folgt die Definition aller OWL Klassen, sprich aller Konzepte oder Muster diverser Entitäten in diesem Wissensbereich. In Abbildung 4 wird die Klasse *Beach* dargestellt, welche eine Unterklasse der Klasse *Destination* ist [vgl. Hitz08, 129-133].

```
<owl:Class rdf:about="http://www.owl-ontologies.com/travel.owl#Beach">
        <rdfs:subClassOf rdf:resource=
                "http://www.owl-ontologies.com/travel.owl#Destination"/>
</owl:Class>
```

Abbildung 4: OWL Klasse [siehe Trav13]

Schlussendlich werden noch alle Individuen definiert. Diese sind konkrete Ausprägungen einer jeweiligen Klasse. In Abbildung 5 wird das Individuum *BondiBeach* dargestellt, es ist vom Typ *Beach* [vgl. Hitz08, 129-130].

```
<owl:NamedIndividual rdf:about=
        "http://www.owl-ontologies.com/travel.owl#BondiBeach">
        <rdf:type rdf:resource=
                "http://www.owl-ontologies.com/travel.owl#Beach"/>
</owl:NamedIndividual>
```

Abbildung 5: OWL Individuum [siehe Trav13]

2.2.2 OWL Versionen

Es gibt drei verschiedene Teilsprachen von OWL. *OWL Full* beinhaltet als einzige Version alle OWL und RDF(S) Sprachelemente. Die einzige Voraussetzung ist, dass das Resultat gültiges RDF ist. Bei *OWL Full* entstehen durch mangelnde Einschränkungen Probleme beim Berechnen von Inferenzen, da diese in bestimmten Fällen unentscheidbar sein können. Daher unterstützt bislang keine Inferenzmaschine *OWL Full* vollständig. *OWL DL* ist eine Teilmenge von *OWL Full*, bei der manche Sprachelemente eingeschränkt sind. Daher ist *OWL DL* immer entscheidbar und hat von den drei Versionen die größte praktische Bedeutung. *OWL Lite* ist wiederum eine Teilmenge mit weiteren Einschränkungen von *OWL DL* und ist in der Praxis von geringer Bedeutung [vgl. Shad06, 98; Hitz08, 151-154].

2.2.3 Ontology Learning

Es wurde bereits einleitend erwähnt, dass das automatisierte Ontology Learning große Vorteile für die Skalierbarkeit und den Erfolg des Semantic Web bringt. Dieser Abschnitt soll einen kurzen Einblick darin geben, wie die automatisierte

Erstellung von Ontologien möglich sein kann. Im Wesentlichen werden Ontologien durch das Ausnutzen bereits vorhandener Webressourcen in einem meist iterativen Prozess modelliert. Die folgenden vier verschiedenen Methoden können dafür zum Beispiel angewandt werden [vgl. MaSt01, 76 f.]:

1. Lexical Entry & Concept Extraction: Bei dieser Basismethode werden Webdokumente gescannt und statistisch analysiert, so wird ein Lexikon von Fachtermini erstellt. Die Begriffe können dann entweder ein neues Konzept bilden oder mit einem bestehenden Konzept verlinkt werden.
2. Hierarchical Concept Clustering: Hierbei werden die Konzepte klassifiziert und in eine Hierarchie beziehungsweise Taxonomie gebracht. Die Einstufung kann anhand der Ähnlichkeit der Eigenschaften der Konzepte erfolgen.
3. Dictionary Parsing: Diese Methode basiert auf der Ausnutzung von sogenannten Machine-readable dictionaries (MRD), die für viele Domänen verfügbar sind (zum Beispiel Glossare für die Erklärung von Fachbegriffen). Diese werden geparst und analysiert, um Zusammenhänge festzustellen, die in einer Ontologie erfasst werden können.
4. Association Rules: Die letzte Methode basiert auf dem Konzept des Data Mining, und versucht, Verbindungen zwischen Elementen beziehungsweise Konzepten auf Taxonomie-Ebene festzustellen. Dies könnte bei der Betrachtung von Produkten eines Supermarkts zum Beispiel die Feststellung sein, dass Snacks und Getränke oft gemeinsam gekauft werden.

2.3 Protégé Ontology Editor

Protégé ist ein freier Open Source Ontologie Editor, welcher vom Institut für Biomedizinische Informatik an der kalifornischen Stanford University entwickelt wurde. Er ist der mit Abstand bekannteste und am weitesten verbreitete Ontologie Editor und ist schon seit vielen Jahren verfügbar. Die erste Version erschien bereits 1987, also lange Zeit bevor der Begriff des Semantic Web definiert wurde und sogar ein paar Jahre bevor überhaupt das World Wide Web seinen Siegeszug antrat. Damals war Protégé eine kleine Applikation, das Ziel war der Aufbau von Tools für die Aneignung von Wissen im Bereich der medizinischen Planung. Heute ist der Protégé Ontology Editor ein mächtiges Werkzeug zur Erstellung, Modifizierung und Visualisierung von Ontologien, und un-

terstützt eine Vielzahl von Repräsentationsformaten wie OWL, RDF(S) und XML Schema. Protégé wurde komplett in Java geschrieben, wodurch er plattformunabhängig ist. Dies ist typisch für den wissenschaftlichen Bereich, können doch so Forscher verteilt über die ganze Welt mit unterschiedlichsten Systemen Protégé ohne Einschränkungen nutzen. Einer der größten Vorteile von Protégé ist seine Erweiterbarkeit. Durch seine gut strukturierte API und seine komplett modulare Architektur ist es einfach, Plugins für Protégé zu entwickeln. Dies beinhält unschätzbares Potential und kann zu deutlichem Mehrwert führen. Hinter Protégé steht eine starke Community bestehend aus Entwicklern und akademischen, öffentlichen aber auch privatwirtschaftlichen Benutzern [vgl. Prot13a; Prot13b; Prot13c; Prot13d; Genn03, 89; Hitz08, 155].

2.3.1 Aufbau der Benutzeroberfläche von Protégé

Abbildung 6 zeigt den standardmäßigen Aufbau der Benutzeroberfläche von Protégé nach dem Laden der Travel Ontologie.

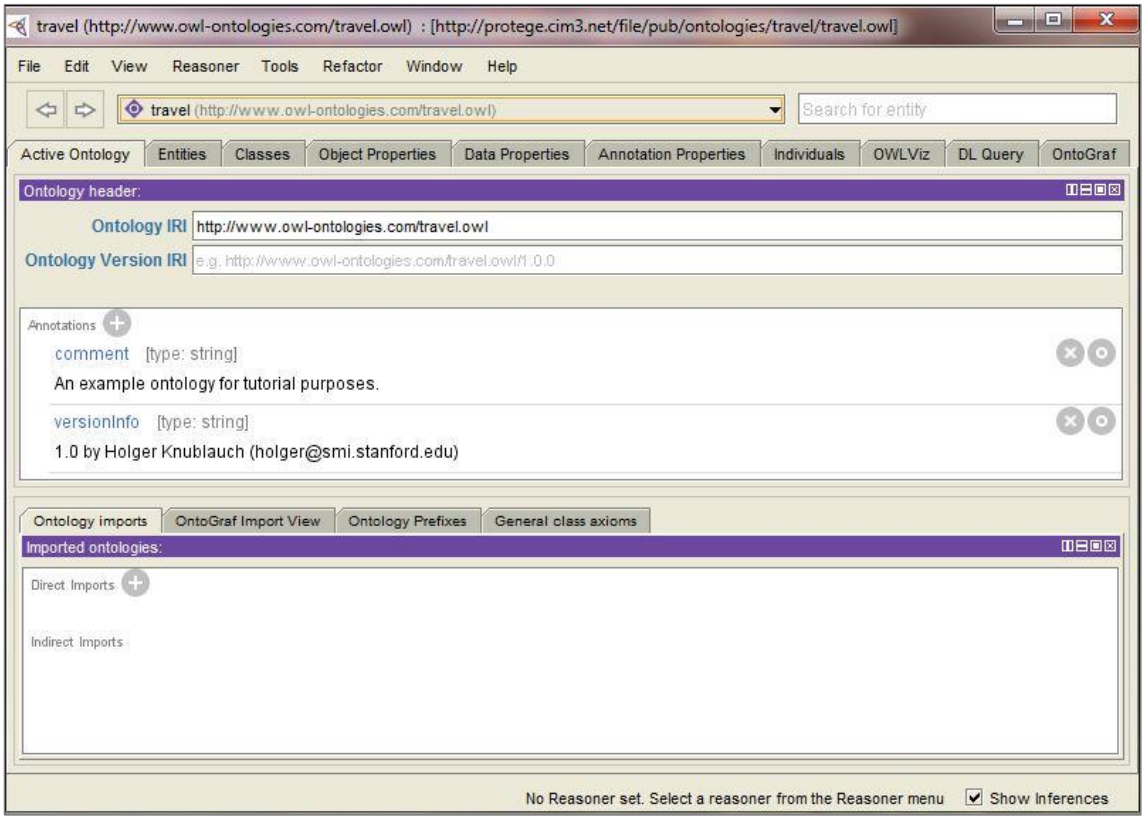


Abbildung 6: Protégé GUI

Diese Ansicht lässt sich flexibel den eigenen Bedürfnissen anpassen. Die Anzeige basiert auf Tabs und Views. Tabs sind die Reiter unter der Adressleiste,

es gibt viele vordefinierte davon. In Abbildung 6 sieht man zum Beispiel unter dem Tab Active Ontology den Kopf einer Ontologie, welcher die Informationen aus der OWL Notation in Abbildung 2 grafisch darstellt. Weitere Tabs gibt es zum Beispiel für OWL Klassen, Individuen, Object und Data Properties und für die grafische Visualisierung mittels OWLViz und OntoGraf. Alle aktuell verfügbaren Tabs lassen sich unter Window → Tabs aktivieren, diese Liste lässt sich durch das Hinzufügen weiterer (auch selbstgeschriebener) Plugins entsprechend erweitern. Views sind die einzelnen Fenster innerhalb eines Tabs, welche sich beliebig nebeneinander anordnen und übereinander stapeln lassen. Über Window → Views lassen sich für jeden Teilbereich einer Ontologie (Klassen, Individuen, Properties, und viele mehr) alle verfügbaren Views auswählen und im aktuellen Tab platzieren. Auch die Menge von Views kann durch das Hinzufügen von Plugins erweitert werden. Gemeinsam mit dem Hauptfenster von Protégé öffnet sich ein zugehöriges Konsolenfenster, in dem Log-Nachrichten ausgegeben werden. Dieses ist zwecks Information und Fehlersuche sehr hilfreich. Über Help → About kann neben der Version von Protégé eine Liste mit allen geladenen Plugins und deren Versionen angezeigt werden.

2.4 Games with a purpose (GWAPs)

Games with a purpose sind Computerspiele, bei denen menschliche Spieler mit ihren Fähigkeiten dabei helfen, Probleme zu lösen, die durch Computer nicht vollständig im Alleingang gelöst werden könnten.

In der heutigen Zeit existiert ein riesiger Computerspielemarkt. Viele Menschen rund um den Globus verbringen im Kollektiv sehr viel Zeit mit Computerspielen. Die Grundidee hinter GWAPs ist die Suche nach einer Möglichkeit, diese verbrachte Zeit mit Computerspielen gezielt produktiv zu nutzen und die gemeinsame Arbeitskraft für etwas Sinnvolles zu kanalisieren. Viele Menschen sollen zur gleichen Zeit unbewusst gemeinsam Probleme in großem Maßstab lösen, die für einzelne Personen nicht bewältigbar wären. Es ist erwiesen, dass das kollektive Wissen einer großen Anzahl von Menschen oft auch zu qualitativ besseren Resultaten als der Einsatz einzelner bereichsspezifischer Experten führt [vgl. Vahn06, 92; RaSc09, 193].

Der Grund, dass solche Probleme auch nicht durch automatisierte Ansätze vollständig gelöst werden können, ist, dass Computer trotz riesiger Fort-

schritte in den letzten Jahrzehnten weiterhin viele Dinge nicht können, die für Menschen trivial sind. Das menschliche Gehirn ermöglicht eine Wahrnehmungsfähigkeit, die für Computer bis heute nicht möglich ist. Jeder Mensch hat ein bestimmtes semantisches Verständnis und Fingerspitzengefühl, welches beim Einsatz von GWAPs genutzt werden soll. Nur durch diesen menschlichen Input können einerseits wie erwähnt bestimmte Probleme gelöst werden, andererseits kann durch Computer versucht werden, bestimmte Muster in diesen Eingaben zu erkennen, damit Computer auch vom Menschen lernen und dadurch „intelligenter“ werden können [vgl. Vahn06, 96].

Über das Internet können Millionen Menschen gleichzeitig gemeinsam ein Problem lösen. Durch GWAPs wird versucht, diesen unschätzbaren Skalierungseffekt gezielt zu nutzen [vgl. Vahn06, 94].

2.4.1 Design von GWAPs

Die grundsätzliche Zielstellung von GWAPs ist es, korrekte Lösungen beziehungsweise nützliche Daten zu erbringen, und gleichzeitig Spaß zu machen [vgl. Vahn06, 94]. Gerade ein umfassendes und effektives Anreizsystem ist eine wichtige Voraussetzung für den Erfolg eines GWAPs, da ansonsten nur wenig Bereitschaft der Benutzer besteht, das GWAP zu spielen. Das wichtigste Ziel hierbei ist das Generieren intrinsischer Motivation. Dadurch wird garantiert, dass der Benutzer viel Zeit mit dem Spiel verbringt. Dies kann durch möglichst gute Unterhaltung beim Spielen, oder durch Auszeichnungen und Preise (zum Beispiel beim Überschreiten bestimmter Punktegrenzen) erreicht werden. Auch durch das Schaffen immer neuer Herausforderungen kann eine langfristige Motivation der Spieler sichergestellt werden. Besonders vielversprechend ist die Integration von GWAPs in soziale Netzwerke, da hier durch die große Anzahl der potentiellen Spieler und den sozialen Kontext durch Wettbewerb oder das Teilen von Ergebnissen mit Freunden eine hohe intrinsische Motivation erreicht werden kann [vgl. RaSc09, 194; Weic11, 1054 f.].

Der zweite wichtige Punkt bei GWAPs ist die Validierung der Resultate. Es ist unerlässlich, eine unbewusste Verfälschung oder sogar bewusste Manipulation der Ergebnisse seitens der Spieler zu vermeiden. Gerade in sozialen Netzwerken besteht die Gefahr, dass Benutzer zusammenarbeiten, um durch Manipulation (zum Beispiel durch Abstimmung der Antworten) möglichst hohe

Punktzahlen oder ähnliches zu erreichen. Die hier erzielten Ergebnisse sind dann unbrauchbar und können im schlimmsten Fall die gesamte Untersuchung verfälschen. Diese Verhaltensweise lässt sich zum Beispiel durch die Verschleierung der Identität der anderen Spieler oder das Vermeiden von Mustern bei der Reihenfolge der Antwortmöglichkeiten verhindern. Auch eine Analyse des Antwortverhaltens der Spieler oder das Einführen von Vertrauensstufen, die die Gewichtung der Antworten einzelner Spieler in der Gesamtuntersuchung beeinflussen, können helfen, Manipulation zu vermeiden [vgl. RaSc09, 193 f.].

Werden diese Designkriterien eingehalten, kann mit Hilfe von GWAPs Wissen unter wissenschaftlichen Kriterien wie Objektivität, Reliabilität und Repräsentativität erfasst werden [vgl. Weic11, 1054].

2.4.2 Beispiele und Anwendungsbereiche für GWAPs

Es gibt zwei Arten von GWAPs: closed-end games und open-end games. Der Unterschied liegt in der Art der Antwortmöglichkeiten. Während bei closed-end games nur zwischen vorgegebenen Antworten ausgewählt werden kann (zum Beispiel Ja oder Nein, einer Skala, oder Multiple Choice), sollen die Spieler bei open-end games komplett eigenständige Antworten geben [vgl. RaSc09, 194].

Ein Beispiel für open-end games ist das ESP Game, eines der allerersten GWAPs. In diesem sollen die Spieler Bilder mit Beschriftungen versehen. Man soll jeweils raten, welche Bezeichnung der Spielpartner wählt. Erraten beide Spieler denselben Begriff, bekommen sie Punkte dafür. Das Spiel zielt darauf ab, Bilder im Internet mit Beschriftungen zu „taggen“, welche für Suchmaschinen bei der Bildersuche verwendet werden können [vgl. Vahn06, 92 f.]. Ein weiteres Beispiel ist Peekaboom, wo es darum geht, Objekte in Bildern manuell zu lokalisieren. Im Hintergrund lernt der Computer mit – es wird hierbei versucht Computer Vision Algorithmen zu trainieren [vgl. Vahn06, 93].

Ein Beispiel für ein closed-end game ist das Sentiment Quiz, welches versucht die Stimmung bestimmter Aussagen oder Wörter zu definieren. Die Skala reicht hierbei in 5 Stufen von sehr negativ bis sehr positiv. [vgl. RaSc09, 193 f.] Die Daten des Sentiment Quiz werden in weiterer Folge dazu verwendet, ein Stimmungs-Lexikon aufzubauen. Dieses soll schlussendlich für bestimmte Anwendungsfälle wie politische Kampagnen oder Marktforschung dienlich sein, welche ein großes Interesse an einer automatisierten Interpretation von Mei-

nungen im Web haben, die auf Basis des erlernten Lexikons durchgeführt werden kann [vgl. Weic11, 1053].

Weitere Anwendungsbereiche können eine automatisierte Sprachübersetzung, das Zusammenfassen von Texten, eine weiter verbesserte Websuche oder „Adult content filtering“ (das Filtern von für Kinder ungeeigneten Inhalten) sein [vgl. Vahn06, 92 ff.].

2.4.3 Relevanz von GWAPs für diese Arbeit

Der Einsatz eines GWAPs hat für die Lösung der Problemstellung und die Beantwortung der Forschungsfrage dieser Arbeit hohe Relevanz. Durch die Auslagerung der Validierung von Ontologien an ein externes GWAP soll gezeigt werden, dass Crowd-Sourcing durch bessere Skalierbarkeit die manuelle Erstellung von Ontologien durch teure Experten ergänzen und vereinfachen kann.

2.5 Java

Java ist eine Programmiersprache und Laufzeitumgebung, die 1995 von Sun Microsystems veröffentlicht wurde [vgl. Java13a]. Sie ist für diese Arbeit vor allem für die Erstellung des Plugins essentiell, da der Protégé Ontology Editor komplett auf Java basiert und alle entsprechenden Plugins daher ebenfalls in Java entwickelt werden. Laut Herstellerangaben ist Java auf 1,1 Milliarden Desktop-PCs, sowie auf 3 Milliarden Mobiltelefonen installiert [vgl. Java13b]. Aber auch auf vielen anderen Geräten wie Blue-ray Disc Playern, Set-Top-Boxen, Druckern, TV-Geräten, Fahrzeug-Navigationssystemen, Geldautomaten, medizinischen Geräten und Parkgebührautomaten wird Java verwendet. Klassische Einsatzszenarien sind auch mobile Anwendungen, Spiele, webbasierter Content und Unternehmenssoftware [vgl. Java13a; Java13b].

2.5.1 Geschichte von Java

Java wurde von Sun Microsystems mit der Zielsetzung entwickelt, eine neue, einfache und handliche objektorientierte Programmiersprache zu kreieren. Ein wichtiges Designkriterium war auch, dass fehlerhafte Java-Programme keinen Schaden im Betriebssystem anrichten können sollen. Die ursprüngliche Version, genannt HotJava, war jedoch nicht besonders erfolgreich, da sie nicht von den Benutzern und Entwicklern angenommen wurde. Erst mit der Lizenzierung

von Java 1995 durch Netscape und die Integration in deren Browser Netscape Navigator begann der Erfolg von Java. 1996 wurde schließlich die erste Version des Java Development Kit (JDK) veröffentlicht, wodurch von diesem Zeitpunkt an jeder Java-Programme entwickeln konnte. Sun Microsystems geriet im Laufe der Jahre in finanzielle Schwierigkeiten und wurde 2009 von der Oracle Corporation übernommen, die seit diesem Zeitpunkt auch Java weiterentwickelt [vgl. Ulle11, 47 ff.; KrHa11, 37-44].

2.5.2 Eigenschaften und Vorteile von Java

Die von Beginn an definierte Zielstellung bei der Entwicklung von Java lautete: „Java soll eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, performante, nebenläufige, dynamische Programmiersprache sein“ [siehe KrHa11, 44]. Diese Ziele wurden weitgehend erreicht und so enthält die Java-Laufzeitumgebung (JRE, Java Runtime Environment) heute die Java Virtual Machine, die Hauptklassen der Java Plattform und weitere umfangreiche Klassenbibliotheken [vgl. Java 13a].

2.5.2.1 Plattformunabhängigkeit

Java ist plattformunabhängig, das heißt Java Programme sind portable Anwendungen, die auf nahezu jeder Rechnerplattform beziehungsweise -architektur ausführbar sind [vgl. Java13b; Orac13]. Die Besonderheit ist hierbei die sogenannte Java Virtual Machine (JVM), die als Vermittler zwischen den Java-Programmen und der Betriebssystem-API fungiert. Diese JVM gibt es in den verschiedensten Versionen für nahezu alle Betriebssysteme und Rechnerarchitekturen. Der große Vorteil hierbei ist, dass bei der Kompilierung von Java-Programmen Java Bytecode für die JVM generiert wird, der nicht an eine Architektur gebunden ist. Die Entwickler nutzen bei der Programmierung also immer die Java-API und nie die tatsächliche API der konkreten Plattform. Die JVM implementiert das Sandbox-Prinzip, indem sie den Bytecode in einer geschützten und kontrollierten Umgebung ausführt [vgl. Ulle11, 50-54].

2.5.2.2 Objektorientierung

Objektorientierte Programmierung ist der Versuch, durch bessere Modellierung die Komplexität der Software-Entwicklung zu entschärfen, und sie so an die

menschliche Denkweise anzupassen. Es gibt Klassen und Objekte, die wie im richtigen Leben Eigenschaften (Attribute) und Funktionen (Methoden) haben. Durch objektorientiere Programmierung werden modulare, wiederverwendbare Softwarekomponenten geschaffen. Java geht bei der Objektorientierung einen Kompromiss und implementiert sie nicht zu 100 Prozent, da auch primitive Datentypen wie Integer (Ganzzahlen) oder Float (Gleitkommazahlen) zugelassen werden [vgl. Ulle11, 54; KrHa11, 44 ff.].

2.5.2.3 Einfachheit

Java hat eine bewusst einfach gehaltene Syntax, die zwar unter Umständen zu etwas mehr Schreibarbeit als bei vergleichbaren Programmiersprachen führt, dafür jedoch übersichtlicher und leichter zu lernen und zu verstehen ist. Zum Beispiel gibt es bei Java keine benutzerdefinierten überladenen Operatoren, um die Verständlichkeit zu verbessern [vgl. Orac13; Ulle11, 60 ff.; KrHa11, 44 ff.].

2.5.2.4 Verbreitung

Ein großer Vorteil von Java, der natürlich auch ein betriebswirtschaftlicher Faktor ist, ist seine weite Verbreitung. Diese wird auch durch die oben erwähnte Plattformunabhängigkeit erreicht. Wichtig hierfür sind die konsistente Laufzeitumgebung auf allen Geräten, sowie die Optimierung auch für Embedded Devices. Es gibt weltweit viele Java-Entwickler, auch die meisten Universitäten lehren Java [vgl. Orac13; Ulle11, 55].

2.5.2.5 Automatisches Speichermanagement

Das automatische Speichermanagement in Java kann vor allem mit zwei Funktionen der Laufzeitumgebung beschrieben werden. Zum einen gibt es in Java keine Zeiger, wie zum Beispiel in C++. Dies verhindert den direkten Zugriff auf beliebige Bereiche im Speicher und somit ein potentielles Sicherheits- und Konsistenzproblem. Verweise auf Objekte in Java funktionieren mit typisierten Referenzen, die in Variablen gespeichert werden. Der Zugriff auf den Speicher ist also für den Benutzer intransparent. Zum anderen gibt es den sogenannten Garbage Collector, der automatisch alle Objekte aus dem Speicher löscht, die nicht mehr referenziert werden. Ein versehentliches exzessives Anfüllen des Speichers ist also nicht möglich [vgl. Ulle11, 57-59; KrHa11, 44 ff.].

2.5.2.6 Ausnahmebehandlung

Java implementiert ein umfangreiches und strukturiertes Exception Handling, welches die Erkennung und kontrollierte Behandlung von Laufzeitfehlern garantieren soll. Eine Methode muss potentielle Exceptions auffangen und behandeln oder sie an ihren Aufrufer weitergeben. Dieser muss sie dann wiederum entweder behandeln oder an seinen Aufrufer weitergeben. So wird sichergestellt, dass jede Ausnahme bearbeitet wird [vgl. Ulle11, 59 f.; KrHa11, 44 ff.].

2.5.2.7 Lizenzierung

Java hatte von Anfang an offenen Quellcode, was die Transparenz und Nachvollziehbarkeit enorm verbessert. Jedoch stand der Code zu Beginn noch unter keiner bestimmten Lizenz. Erst 2007 wurde das Java Development Kit als Open JDK unter die GNU General Public License (GPL) gestellt. Dieses enthielt jedoch einen marginalen Teil an Bibliotheken, welche nicht mit diesem Lizenzmodell vereinbar waren. Diese wurden jedoch im Laufe der Zeit durch passende Bibliotheken ersetzt, sodass das Open JDK heute komplett unter der GPL steht. Die heute häufiger benutzte Version ist jedoch seit der Übernahme von Sun durch Oracle das Oracle JDK, welches die gleiche Basis wie das Open JDK hat und auch zu etwa 95 Prozent gleich ist. Jedoch sind die restlichen Bibliotheken proprietär, sodass das Oracle JDK unter der Binary Code License steht [vgl. Ulle11, 62 ff.].

2.5.3 Limitationen von Java

Java ist eine Sprache zur Lösung allgemeiner Probleme. Überall dort, wo maschinen- oder plattformabhängige Tasks benötigt werden, wird es mit Java problematisch. Dazu zählen hardwarenahe Aufgaben wie das Auswerfen einer CD, der Zugriff auf USB oder der Zugriff auf niedrige Netzwerkprotokolle wie ICMP. Außerdem ist Java nicht für die Programmierung von Treibersoftware geeignet [vgl. Ulle11, 64 f.].

Weiters erlaubt Java per Design manche Funktionalitäten nicht, die in anderen Programmiersprachen gängig sind. Dabei handelt es sich zum Beispiel um die Verwendung von Zeigern, das Überladen von Operatoren oder die Implementierung von Mehrfachvererbung (letzteres ist zumindest rudimentär durch die Verwendung von Interfaces möglich). Der Ausschluss dieser Konzep-

te ist durchaus bewusst und der Zielstellung der Einfachheit und Sicherheit geschuldet [vgl. Ulle11, 64 f.; KrHa11, 44 ff.].

2.6 JSON

JSON ist die Abkürzung für JavaScript Object Notation und ist ein schlankes Datenaustauschformat, das seit 2001 von Douglas Crockford entwickelt wurde. Das Ziel war eine größtmögliche Einfachheit und Vermeidung von unnötiger Komplexität beim Austausch von Daten. So ist JSON sowohl sehr einfach für Menschen zu lesen, als auch für Maschinen zu parsen und zu generieren. Wie der Name bereits andeutet, basiert JSON auf einer Untermenge von JavaScript, wodurch jedes valide JSON-Dokument auch gültiges JavaScript ist. JSON ist als Format unabhängig von Programmiersprachen und daher als Universalkonzept anzusehen. Allerdings übernimmt JSON viele Konventionen aus C-basierten Sprachen und wirkt so für viele Programmierer von Beginn an sehr vertraut. Dennoch bleibt JSON ein zwischen verschiedensten Programmiersprachen unabhängig austauschbares Datenformat. JSON wurde 2006 offiziell im RFC 4627 spezifiziert und formalisiert. Seitdem gibt es auch einen eigenen MIME Type für die Angabe des Medientyps in der Kommunikation mit Webservern über HTTP, welcher *application/json* lautet. JSON wird hauptsächlich für den Datenaustausch zwischen Client und Server verwendet und überall dort, wo eine kompakte und ressourcenschonende Alternative zu XML benötigt wird. Es gibt unzählige Implementierungen von JSON für so gut wie alle gängigen Programmiersprachen. Alleine für Java gibt es über 20 verschiedene JSON-Bibliotheken, wodurch der Einsatz von JSON im Protégé Plugin einfach möglich war [vgl. Json13a; Json13b].

2.6.1 Datenstruktur von JSON

JSON basiert komplett auf dem Unicode, neben UTF-8 sind auch UTF-16 und UTF-32 möglich. Im Prinzip besteht die Struktur von JSON fast ausschließlich aus einer ungeordneten Menge von Paaren von Namen und Werten, allerdings sind auch geordnete Listen von Werten, sogenannte Arrays, möglich.

Objekte bestehen aus einer ungeordneten Menge von Name/Wert-Paaren. Ihr Beginn und Ende werden mit einer geschwungenen Klammer gekennzeichnet. Die Paare bestehen jeweils aus einer Abfolge von Name, Dop-

pelpunkt und Wert; getrennt werden sie durch dazwischenliegende Beistriche. Ein Array ist eine geordnete Liste von Werten, die mit einer eckigen Klammer beginnt und endet. Die Werte dazwischen werden ebenfalls mit Beistrichen getrennt. Ein Wert kann entweder ein Objekt, ein Array, ein String, ein boolescher Wert (true oder false) oder null sein. Dabei können Strukturen beliebig ineinander verschachtelt werden. Ein String ist eine Zeichenkette, die in doppelten Anführungszeichen eingeschlossen ist. Er kann auch sogenannte Escape-Sequenzen wie zum Beispiel \n (neue Zeile) beinhalten. Zwischen den einzelnen JSON-Elementen können beliebig viele Leerzeichen vorhanden sein.

Hier ein Beispiel für ein JSON-Dokument, das eine Person beschreibt:

{ „Vorname“ : „Max“,
„Nachname“ : „Mustermann“,
„Alter“ : 20,
„Hobbies“ : [„Sport“, „Musik“, „Literatur“] }

[vgl. Json13a; Json13b]

2.6.2 Vergleich mit XML

Die Extensible Markup Language (XML) ist weitaus umfangreicher und auch verbreiteter als JSON. Als Auszeichnungssprache versieht sie ihre Daten mit sogenannten Tags. JSON ist hingegen ein reines Datenaustauschformat und ist schlanker und einfacher als XML. Zum Beispiel werden netto weniger Zeichen für die Darstellung der Daten bei JSON benötigt, da der Overhead durch die Tags entfällt. Außerdem ist JSON effizienter zu parsen als XML, was zu einer signifikant höheren Performance führt. Dadurch ist JSON schneller und ressourcenschonender als XML [vgl. Json13b; Nurs09, 157 f.].

2.6.3 Einsatz von JSON in dieser Arbeit

JSON wurde bei der Programmierung des Protégé Plugins für die Kommunikation zwischen Plugin und GWAP im klassischen Client/Server-Modell verwendet. Über JSON werden für die Validierung der Ontologien notwendige Metadaten zwischen den beiden Applikationen kommuniziert. Dies beinhaltet sowohl notwendige Spielparameter wie Typ der Validierung oder Anzahl der notwendigen Antworten, als auch entsprechende Statusinformationen vom GWAP, wie zum Beispiel, ob ein Task erfolgreich abgeschlossen wurde.

3. Entwicklung des Plugins

3.1 Methode und Prototyp-Entwicklung (Pilotstudie)

Die Methode der Arbeit ist durch die Programmierung des Plugins für den Protégé Ontology Editor definiert. Dieser wurde komplett in der plattformunabhängigen Programmiersprache Java geschrieben, deswegen basiert auch die Plugin-Entwicklung auf Java. Für die Programmierung wird das aktuelle Java Development Kit (JDK) benötigt, welches unter anderem den Java Compiler enthält, der den Source Code in Java Byte Code kompiliert, welcher dann von der Java Virtual Machine (JVM) ausgeführt werden kann.

Um die Machbarkeit des Plugins zu demonstrieren, wurde in einem ersten Schritt die Pilotstudie durchgeführt, die sich hauptsächlich mit der Entwicklung eines funktionsfähigen Prototyps auseinander gesetzt hat. Die folgenden drei Abschnitte 3.1.1, 3.1.2 und 3.1.3 beschreiben, wie im Allgemeinen ein funktionierendes Plugin für Protégé erstellt werden kann. Zuerst wird erklärt, wie eine Entwicklungsumgebung eingerichtet werden kann, danach werden der konkrete Aufbau und die konkrete Entwicklung eines Plugins beschrieben. Schlussendlich wird die Programmierung von GUIs (Graphical User Interfaces, Benutzeroberflächen) für Protégé Plugins näher erläutert. Zur Erklärung wird des Öfteren auf den tatsächlichen Prototyp zurückgegriffen. Darüber hinausgehende detailliertere Beschreibungen zum finalen Plugin finden sich im entsprechenden Kapitel 3.2.

3.1.1 Entwicklungsumgebung

Prinzipiell wird für die Erstellung eines lauffähigen Java-Programms nur ein Texteditor und der Java Compiler benötigt. Allerdings ist die Verwendung einer sogenannten integrierten Entwicklungsumgebung (IDE, integrated development environment) sehr empfohlen, da dies viele Vorteile bietet. Die Auswahl einer Umgebung bleibt dem Programmierer und seinen Vorlieben beziehungsweise Anforderungen überlassen. In dieser konkreten Arbeit hat sich der Autor für die Verwendung von Eclipse entschieden, da es unter anderem folgende Vorteile bietet:

- Übersichtliche Projektstruktur im Workspace, gleichzeitiges Bearbeiten aller Dateien aller geöffneten Projekte möglich.
- Nachvollziehbares Einbinden aller benötigten Programmbibliotheken.
- Guter Editor mit Syntax Highlighting, welcher automatisch für fokussierte Code-Ausschnitte Hilfetexte aus der Java Dokumentation (Javadoc) anzeigt. Ist für eine Bibliothek eines Drittherstellers standardmäßig keine Javadoc integriert, lässt sie sich ganz einfach im Nachhinein einbinden.
- Automatische und umfangreiche Fehlererkennung (zum Beispiel für fehlendes Exception Handling), auch schon vor der Kompilierung. Auch Warnungen werden angezeigt, zum Beispiel für nicht verwendete Variablen. Dadurch lässt sich ein Programm schlank und übersichtlich halten.
- Automatische Kompilierung beim Speichern des Projekts.

3.1.1.1 Einrichten der Entwicklungsumgebung

Wie bereits erwähnt, gibt es mehrere Entwicklungsumgebungen zur Auswahl. Selbst bei der Entscheidung für eine bestimmte IDE gibt es mehrere Möglichkeiten, diese den eigenen Anforderungen entsprechend anzupassen. Die folgende ausgetestete Vorgehensweise für die Einrichtung von Eclipse dient also exemplarisch nur als einer von mehreren Wegen, um eine Entwicklungsumgebung für die Programmierung von Plugins für Protégé einzurichten:

1. Download und Installation des aktuellen 32-Bit JDK (enthält auch JRE) von http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.
2. Umgebungsvariablen des Systems aktualisieren:
    a. Zur Variable *Path* den bin-Ordner der JDK-Installation hinzufügen (unter Windows z.B.: „C:\Program Files\Java\jdk1.7.0_21\bin“).
    b. Die Variable *Classpath* wenn noch nicht vorhanden erstellen und „.;..“ hinzufügen (ohne Anführungszeichen). Der Punkt steht für das aktuelle, die zwei Punkte für das übergeordnete Verzeichnis.
    c. Die Variable *Java_Home* wenn noch nicht vorhanden erstellen und den Pfad zum Hauptordner der JDK-Installation hinzufügen (unter Windows z.B. „C:\Program Files\Java\jdk1.7.0_21“).

3. Download und Installation von Protégé in der aktuellsten 32-Bit Version (derzeit Version 4.3) ohne Java VM (da diese schon installiert ist) von http://protege.stanford.edu/download/protege/4.3/installanywhere/Web_Installers/. Bei der Installation aus Stabilitätsgründen die installierte java.exe im bin-Ordner vom JDK auswählen, da diese bei einem „normalen“ Java Update (JRE, nicht JDK) unverändert bleibt.
4. Unter Windows muss in den NTFS-Berechtigungen des plugins-Ordners von Protégé (z.B. „C:\Program Files\Protege_4.3\plugins“) dem Sicherheitsprinzipal „Jeder“ Vollzugriff gegeben werden. Dadurch können Plugin-Updates direkt über Protégé in diesen Ordner installiert werden, ansonsten würden sie in das jeweilige Benutzerverzeichnis installiert.
5. Optional: Download und Installation von Graphviz in der aktuellsten Version von http://www.graphviz.org/Download..php. Angabe des Installationspfads (z.B. „C:\Program Files\Graphviz2.30\bin\dot.exe“) in den Optionen von Protégé unter File → Preferences → OWLViz. Dies dient zur graphischen Darstellung von Ontologien in Protégé im Tab OWLViz, ansonsten tritt bei der Auswahl dieses Tabs eine Runtime Exception auf.
6. Download von Eclipse in der aktuellsten Classic Edition (32-Bit) von http://www.eclipse.org/downloads/ und Entpackung an einem geeigneten Ort (unter Windows z.B.: „C:\Program Files“). Als Workspace einen geeigneten Pfad auswählen und in den Optionen unter Window → Preferences → Java → Installed JREs prüfen, ob die aktuellste JRE ausgewählt ist. Dies darf entweder die „normale“ JRE oder die mitgelieferte JRE im JDK sein.

3.1.2 Erstellung eines Protégé Plugins

In diesem Abschnitt wird die generelle Vorgehensweise bei der Erstellung eines Protégé Plugins beschrieben. Zur Veranschaulichung werden Beispiele aus dem tatsächlichen Prototyp des GWAP Plugins gezeigt. Es wird jedoch versucht, den Vorgang so allgemein wie möglich zu beschreiben, um ihn leicht auf andere Plugin-Entwicklungen umzulegen.

Es gibt grundsätzlich mehrere Arten von Plugins für Protégé. Die gängigste Variante, die auch für das GWAP Plugin verwendet wurde, ist eine sogenannte ViewComponent. Eine View entspricht einem eigenen Fenster in Protégé, in dem abhängig vom aktuellen Zustand (zum Beispiel ausgewählter

Teil der Ontologie) verschiedene Informationen dargestellt und bearbeitet werden können. Es gibt für alle Bereiche einer Ontologie eigene ViewComponents, unter anderem die für diese Arbeit relevanten ClassViewComponents (für OWL Klassen) und IndividualViewComponents (für OWL Individuen).

Zur besseren Veranschaulichung der Zielstellung eines simplen ViewComponent Plugins für Protégé wird die Oberfläche des Prototyps in Abbildung 7 dargestellt. Die GUI besteht aus 3 Labels, 3 Textfeldern und einem Button. Im ersten Textfeld wird die zu validierende OWL Klasse in Abhängigkeit der aktuellen Auswahl automatisch dargestellt. In den beiden anderen Textfeldern werden die Domain, gegen die validiert werden soll, und zusätzliche Informationen für die Validierenden eingegeben. Beim Klick auf den Button werden alle Informationen aus den Textfeldern in einer Message Box ausgegeben.

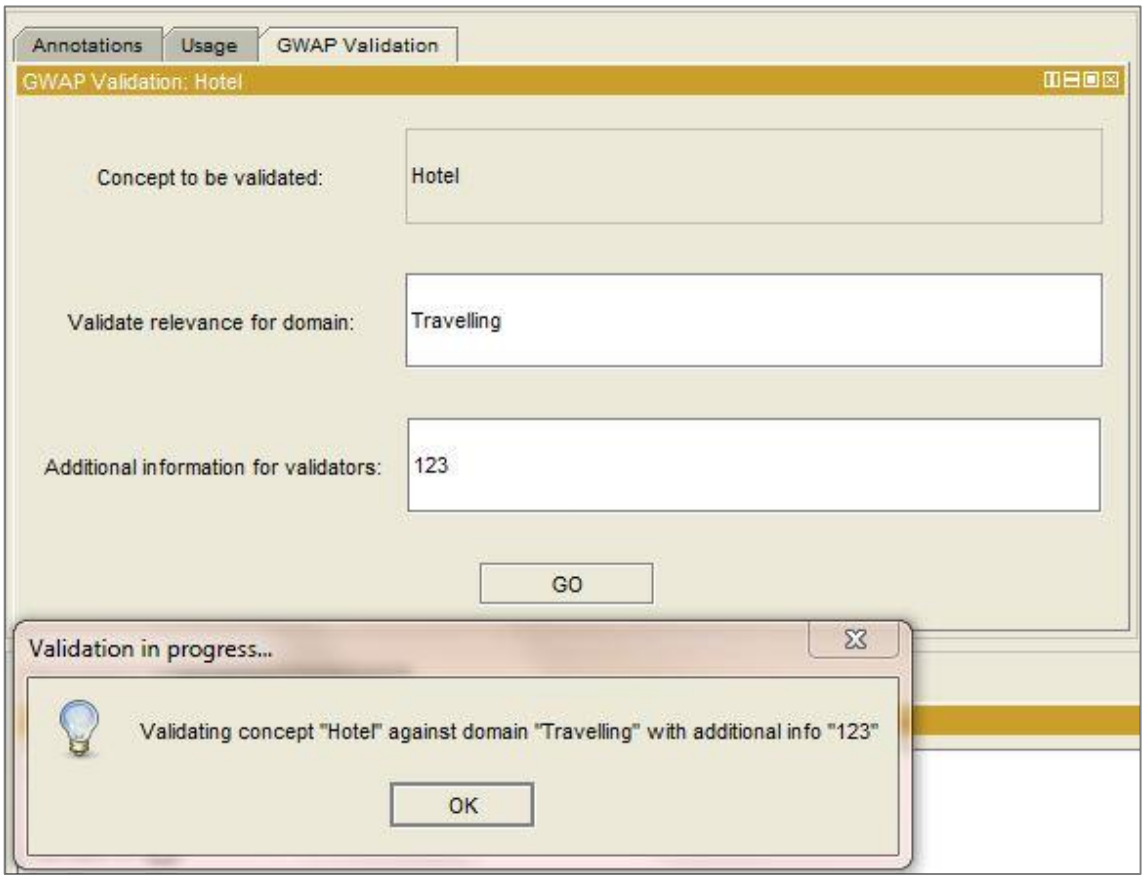


Abbildung 7: Protégé Plugin Prototyp: GUI

Um diese Zielstellung, ein funktionsfähiges Plugin für Protégé zu erstellen, zu erreichen, ist eine Reihe von Schritten notwendig. Um diese besser zu veranschaulichen und die einzelnen Komponenten eines Plugins besser in Bezie-

hung zueinander setzen zu können, wird in Abbildung 8 die Projektstruktur des Prototyps in Eclipse dargestellt.

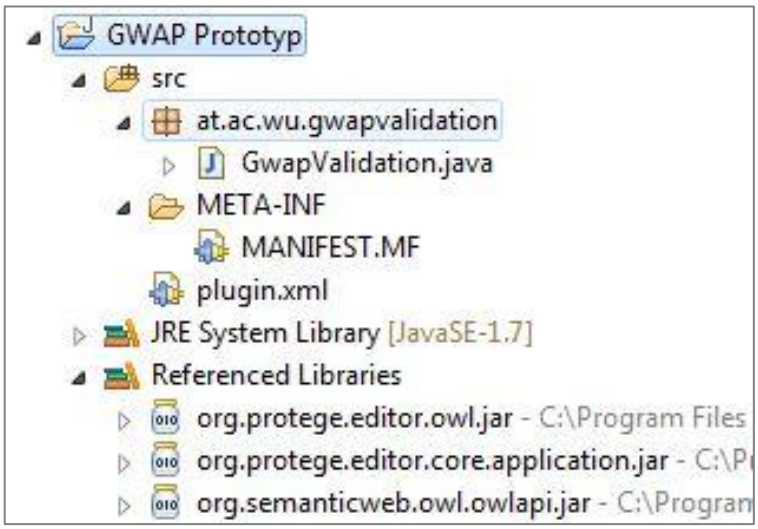


Abbildung 8: Protégé Plugin Prototyp: Eclipse Projektstruktur

Wie leicht zu erkennen ist, werden alle relevanten Dateien im src-Ordner des Projekts abgelegt. Dazu zählen alle Java Klassen, welche wiederum in einem Paket zusammengefasst werden sollten (siehe at.ac.wu.gwapvalidation in Abbildung 8). Für die Entwicklung werden mindestens 3 externe Java-Bibliotheken benötigt (siehe Referenced Libraries in Abbildung 8). Diese befinden sich im Protégé Installationsverzeichnis in den Ordnern bundles und plugins. Das Einbinden funktioniert folgendermaßen: Rechtsklick auf den Projektordner → Properties → Java Build Path. Hier werden alle benötigten Bibliotheken als „External JARs“ hinzugefügt. Sehr hilfreich ist auch das Einbinden der jeweiligen Java Dokumentation (Javadoc). Diese kann über einen Rechtsklick auf die jeweilige Bibliothek → Properties → Javadoc Location angegeben werden. Die Javadoc der Kernklassen von Protégé findet man zum Beispiel im Web unter http://protege.stanford.edu/protege/4.3/docs/api/, die Javadoc der OWL API unter http://owlapi.sourceforge.net/javadoc/. Neben den Java Klassen werden für ein funktionsfähiges Plugin noch eine Manifest-Datei (MANIFEST.MF) und eine Plugin-Datei (plugin.xml) benötigt, in den folgenden Unterabschnitten 3.1.2.1, 3.1.2.2 und 3.1.2.3 werden diese 3 Dateitypen näher beschrieben.

Ein Protégé Plugin ist im Endeffekt nichts anderes als ein Java-Archiv (jar-Datei). Ist man mit der Entwicklung fertig, exportiert man das Plugin daher mittels Rechtsklick auf den src-Ordner → Export → Java → JAR file. Nach ei-

nem Klick auf Next muss man im Baum unter src den gesamten META-INF Ordner inklusive dem Manifest abwählen, da es sonst zu Problemen bei der Einbindung in Protégé kommen kann. Weiters gibt man den Exportpfad an. Nach 2 weiteren Klicks auf Next wählt man als Manifest noch die MANIFEST.MF-Datei aus dem META-INF-Ordner aus und schließt dann mit einem Klick auf Finish den Exportvorgang ab. Nun wird das exportierte jar-File in den plugins-Ordner im Protégé-Verzeichnis kopiert und dann Protégé gestartet. Im Konsolenfenster von Protégé erkennt man, ob das Plugin erfolgreich eingebunden werden kann. Tritt kein Fehler auf, kann man die programmierte View unter Window → Views → Class views (könnte je nach Programmierung natürlich auch ein anderer Typ sein) auswählen und entweder in einem eigenen Bereich ablegen oder die Ansicht auf eine andere View als Tab „stacken“, wie dies auch in Abbildung 7 der Fall ist. Dafür fokussiert man die Mitte der anderen View, bis der Rahmen die ganze View markiert und klickt dann mit der linken Maustaste.

3.1.2.1 Java Klassen

Im Prinzip hat man bei der Programmierung der Java Klassen alle Möglichkeiten der Programmiersprache Java zur Verfügung. Damit das Plugin in Protégé jedoch eingebunden werden kann, muss jede Klasse, die eine ViewComponent sein soll, von einer entsprechenden Klasse von Protégé erben. Im Falle des Prototyps ist das die Klasse *AbstractOWLClassViewComponent*. Da dies, wie der Name schon sagt, eine abstrakte Klasse ist, müssen in der abgeleiteten Klasse alle abstrakten Methoden implementiert werden. In diesem Fall sind das die Methoden *initialiseClassView()* (wird beim Erzeugen der ViewComponent aufgerufen, daher sollten hier alle initialen Tasks wie etwa der GUI-Aufbau stattfinden), *updateView(OWLClass selectedClass)* (wird bei jeder Änderung wie zum Beispiel Selektion einer anderen OWL Klasse aufgerufen, erhält diese daher als Parameter), und *disposeView()* (wird beim Schließen der ViewComponent aufgerufen, eventuelle abschließende Tasks sollten hier durchgeführt werden). Diese Funktionsweise gilt mit kleinen Unterschieden auch für alle anderen ViewComponents wie zum Beispiel die *AbstractOWLIndividualViewComponent*.

3.1.2.2 Manifest Datei

In Java-Archiven können zusätzlich zum eigentlichen Source Code sogenannte Manifest-Dateien vorhanden sein, die ergänzende Meta-Informationen über das Archiv bereitstellen. Ein Manifest ist eine einfache Textdatei, die Name/Wert-Paare enthält. Sie wird in einem Java-Archiv immer mit dem Namen *MANIFEST.MF* im Ordner *META-INF* bereitgestellt [vgl. Ulle11, 1273].

Da Manifest-Dateien in Protégé Plugins zwingend vorgeschrieben sind, weil sie für die Einbindung in Protégé notwendige Informationen enthalten, wird nun der Aufbau einer solchen Datei exemplarisch an der Manifest-Datei des Prototypen erklärt (siehe Abbildung 9). In der folgenden Liste wird für jedes benötigte Attribut der Manifest-Datei (nicht alle überhaupt möglichen Attribute werden für ein Protégé Plugin benötigt) eine kurze Beschreibung angegeben:

- *Manifest-Version:* Version des Manifests, im Normalfall der Wert 1.0.
- *Created-By:* Name des Autors.
- *Bundle-ManifestVersion:* Version des OSGi-Bundle Manifests, meistens 2.

```
Manifest-Version: 1.0
Created-By: Florian Hanika
Bundle-ManifestVersion: 2
Bundle-Name: GWAP Validation
Bundle-SymbolicName: at.ac.wu.gwapvalidation;singleton:=true
Bundle-Description: A plugin for Protege which validates a
 part of an ontology with the help of a GWAP
Bundle-Vendor: Vienna University of Economics and Business
Bundle-ClassPath: .
Import-Package: javax.swing
Bundle-Version: 1.0
Bundle-Activator: org.protege.editor.core.plugin.DefaultPluginActivator
Require-Bundle: org.protege.editor.core.application,org.protege.editor.owl,
 org.semanticweb.owl.owlapi
Built-By: Florian Hanika
Build-Date: 16/03/2013
```

Abbildung 9: Protégé Plugin Prototyp: Manifest Datei

- *Bundle-Name:* Name des Bundles (modularisierte Softwarekomponente), sollte sinnvoller Weise dem Plugin-Namen entsprechen.
- *Bundle-SymbolicName:* Symbolischer Name des Bundles, entspricht dem Paketnamen. Dieser eindeutige Name ist äußerst wichtig, da das Plugin über diesen in Protégé eingebunden wird. Die Zusatzoption *singleton:=true* gibt an, dass nur ein Bundle mit exakt diesem Namen im Framework (ent-

spricht in diesem Fall allen Protégé Paketen inklusive alle Plugins) vorkommen darf. Es kann also nicht passieren, dass zwei verschiedene Versionen des Plugins gleichzeitig in Protégé eingebunden werden.

- *Bundle-Description:* Textbeschreibung des Bundles/Plugins.
- *Bundle-Vendor:* Hersteller des Bundles, zum Beispiel Universität/Firma.
- *Bundle-Classpath:* Java-Classpath des Bundles, im Normalfall nur ein Punkt (.), der für das jeweils aktuelle Verzeichnis steht.
- *Import-Package:* Alle Pakete, die für das Funktionieren des Plugins importiert werden müssen, in diesem Fall alle benötigten Standard Java-Pakete.
- *Bundle-Version:* Version des Bundles/Plugins. Diese ist in Kombination mit dem Symbolic Name der eindeutige Identifier des Bundles.
- *Bundle-Activator:* Protégé Klasse, die das Plugin lädt.
- *Require-Bundle:* Alle anderen Bundles/Plugins, die geladen werden müssen, damit das Plugin funktioniert. Im Gegensatz zur Option „Import-Package“ sind dies alle externen Nicht-Standard Java-Pakete.
- *Built-By:* Name des Builders (kann dem Autor-Namen entsprechen).
- *Build-Date:* Datum des Builds.

3.1.2.3 Plugin Datei

Für das korrekte Einbinden des Plugins in Protégé wird weiters eine Datei namens *plugin.xml* benötigt. Diese befindet sich im Java-Archiv auf oberster Verzeichnis-Ebene und beschreibt, wo in Protégé welche Java-Klassen aus dem Plugin auf welche Art und Weise eingebunden werden. Zur Veranschaulichung wird wiederum die Plugin-Datei des Prototyps vorgestellt (siehe Abbildung 10).

```
<?xml version="1.0" ?>
<plugin>
  <extension id="at.ac.wu.gwapvalidation.GwapValidation"
    point="org.protege.editor.core.application.ViewComponent">
      <label value="GWAP Validation"/>
      <class value="at.ac.wu.gwapvalidation.GwapValidation"/>
      <headerColor value="@org.protege.classcolor"/>
      <category value="@org.protege.classcategory"/>
  </extension>
</plugin>
```

Abbildung 10: Protégé Plugin Prototyp: Plugin Datei

Im Folgenden wird die Bedeutung der einzelnen Informationen in den XML-Tags erläutert:

- *extension id:* Eine eindeutige ID für das Plugin im Framework, hierfür wird der vollqualifizierte Klassenname inklusive dem Paketpfad verwendet.
- *extension point:* Der Extension Point gibt an, was das Plugin machen möchte. In diesem Fall soll mit dem Plugin die Funktionalität einer ViewComponent implementiert werden.
- *label value:* Der Name beziehungsweise die Beschriftung, mit der das Plugin in Protégé bei der Auswahl im Menü aufscheint.
- *class value:* Der Class Value gibt den vollqualifizierten Klassennamen inklusive Paketpfad der Plugin-Klasse an, die eingebunden werden soll.
- *headerColor value:* Die Farbe, in der der Header (Titelleiste) des Fensters des Plugins in Protégé aufscheint. Hier werden Konstanten hinterlegt, in diesem Fall der Standard-Wert für eine ClassView.
- *category value:* Die Kategorie, unter der das Plugin in Protégé im Menü bei der Auswahl der Views unter Window → Views aufscheint. Auch hier werden Konstanten hinterlegt, in diesem Fall wiederum der Standard-Wert für ClassViews.

3.1.3 GUIs in Protégé Plugins

An dieser Stelle soll ausführlich auf die Programmierung von GUIs für Protégé Plugins eingegangen werden, da dies entgegen aller Erwartungen eine der größten Herausforderungen bei der Entwicklung des Prototyps war.

Wie bereits erwähnt findet der Aufbau der GUI (siehe Abbildung 7) in der Initialisierungsmethode der ViewComponent statt, zum Beispiel bei ClassViews in der Methode *initialiseClassView()*. Das Problem hierbei ist nun, dass in dieser Methode die Größe der Component beziehungsweise des Java Containers noch nicht abgerufen werden kann. Beim Aufruf der entsprechenden Methode *getSize()* bekommt man für beide Achsen jeweils den Wert 0 zurück. Dies liegt daran, dass zu diesem Zeitpunkt die GUI noch nicht berechnet und dargestellt wurde. Normalerweise kann man sich in Java in einem solchen Fall über die Standardmethoden *validate()* beziehungsweise *revalidate()* oder *doLayout()* behelfen, um die Berechnung zu forcieren, in diesem Fall funktioniert das jedoch nicht. Auch ein Erzwingen der Sichtbarkeit der GUI mittels *setVisible(true)* schafft keine Abhilfe, bei keiner Kombination der erwähnten Methoden erhält man die gewünschten Werte. Dies ist insofern problematisch, weil dadurch das

ohnehin schon sehr komplexe GridBagLayout, mit dem eine beliebige Anordnung der GUI-Elemente möglich wäre, nicht mehr anwendbar ist. Beim Vergrößern oder Verkleinern der Ansicht verschieben sich alle Elemente, da keine Anpassung auf die tatsächliche aktuelle Größe der Component aufgrund oben genannter Schwierigkeiten möglich ist.

Abhilfe schafft das sogenannte BoxLayout, mit dem Boxen (logische Zusammenschlüsse von GUI-Elementen) beliebig vertikal oder horizontal gestapelt werden können. Bei optimalem Einsatz ist dieses Layout fast so mächtig wie das GridBagLayout, mit dem großen Vorteil, dass relative Zuordnungen zwischen den Elementen geschaffen werden können, die unabhängig von der Gesamtgröße der Component sind. Für den erfolgreichen Einsatz sind jedoch ein paar Besonderheiten zu beachten. Jedem Java-Programmierer, der sich intensiver mit GUIs auseinander setzt, ist bewusst, dass die Darstellung eines Elements von verschiedenen Größen abhängt, nämlich von seiner *Minimum*, *Maximum* und *Preferred Size*. Leider ist das Verhalten auch von Element zu Element unterschiedlich, Textfelder zum Beispiel ändern ihre Größe beim Resizen standardmäßig anders als Labels, nämlich abhängig vom aktuellen Inhalt. Dies schafft gerade bei der Kombination dieser beiden Elemente in einer Zeile Probleme, die man folgendermaßen umgehen kann:

Einerseits muss die MaximumSize aller Elemente richtig gesetzt werden. Die tatsächliche Größe ist hierbei egal und kann auch viel größer als die Bildschirmgröße sein. Wichtig ist nur das Verhältnis, will man zum Beispiel wie in Abbildung 7 ein Textfeld doppelt so breit wie ein Label machen, muss auch die MaximumSize des Textfelds doppelt so groß wie die des Labels sein. Andererseits muss auch die PreferredSize aller Elemente im richtigen Verhältnis gesetzt werden. Bei Textfeldern funktioniert dies laut der Erfahrung des Autors nur über die Methode *setColumns(columns)*, wobei die Minimalgröße eines Textfelds 6px beträgt, und für jede Spalte 8px dazu addiert werden. Will man die PreferredSize des Textfelds *fConcept* also wiederum doppelt so groß setzten wie die des Labels *lConcept*, ist dies mit folgendem Methodenaufruf möglich:
*fConcept.setColumns((int) (((lConcept.getPreferredSize().width * 2) - 6) / 8));*
Setzt man die PreferredSize eines Textfelds überhaupt nicht, verändert das Textfeld seine Breite beim Resizen abhängig vom Inhalt, was zu unvorhersehbaren Problemen bei der Darstellung führen kann.

3.2 Beschreibung des Plugins

3.2.1 Funktionsumfang des Plugins

Das finale Plugin besteht aus zwei Teilkomponenten. Es ist einerseits die Validierung von Konzepten (OWL Klassen), als auch die Validierung von Individuen (OWL Individuals) möglich. Beide Komponenten sind als entsprechende ViewComponents realisiert und werden als ein Protégé Plugin in einem jar-File zusammengefasst. Um alle Funktionen und Möglichkeiten des Plugins zu erläutern, betrachten wir zunächst Abbildung 11 für die Validierung von Konzepten.

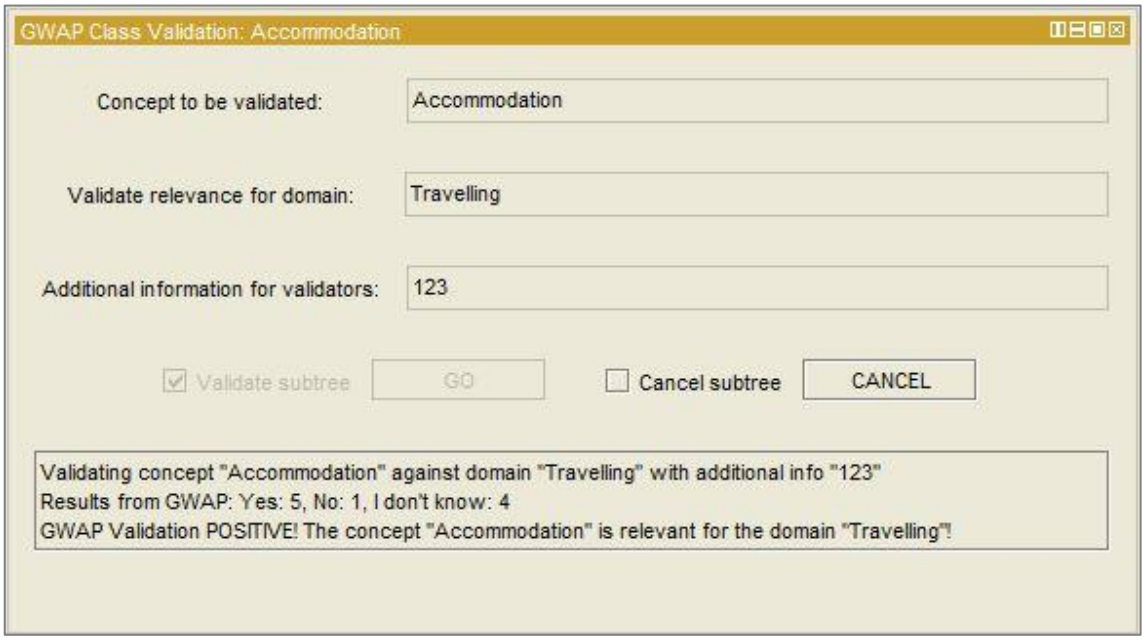


Abbildung 11: Protégé Plugin: GUI Konzepte

Man kann erkennen, dass die GUI im Vergleich zum Prototyp deutlich erweitert wurde. Nach wie vor gibt es drei Labels und drei Textfelder, in letzteren werden die Daten für die Validierung erfasst. Neben dem GO-Button befindet sich nun eine Checkbox, mit der man die Validierung des kompletten Teilbaums aktivieren kann. Hat das Konzept Accommodation weitere OWL Klassen als Subklassen (zum Beispiel Hotel oder Campground), wird für diese bei aktivierter Checkbox beim Klicken des GO-Buttons ebenfalls eine Validierung gestartet. Es gibt nun weiters einen CANCEL-Button, mit dem eine laufende Validierung abgebrochen werden kann. Dieser wird ebenfalls durch eine Checkbox für den Abbruch des ganzen Subtrees ergänzt. Direkt darunter befindet sich eine Text Area für die Ausgabe aller Statusinformationen. Hier werden Nachrichten beim Starten, beim Abbrechen und beim Vorliegen eines Ergebnisses einer Validie-

rung angezeigt. Auch entsprechende Fehlermeldungen werden ausgegeben, zum Beispiel wenn ein Fehler bei der Kommunikation mit dem GWAP auftritt. In Abbildung 11 wurde die Validierung des Konzepts positiv abgeschlossen, da für mehr Leute der Zusammenhang zwischen Konzept und Domäne gegeben war als umgekehrt. Bei Gleichstand wäre das Ergebnis undefiniert und wenn die Mehrheit keinen Zusammenhang sieht, ist das Ergebnis negativ. In letzterem Fall erscheint unter der Text Area noch ein Button, mit dem das Konzept aus der Ontologie gelöscht werden kann.

Betrachten wir nun Abbildung 12 für die Validierung von Individuen:

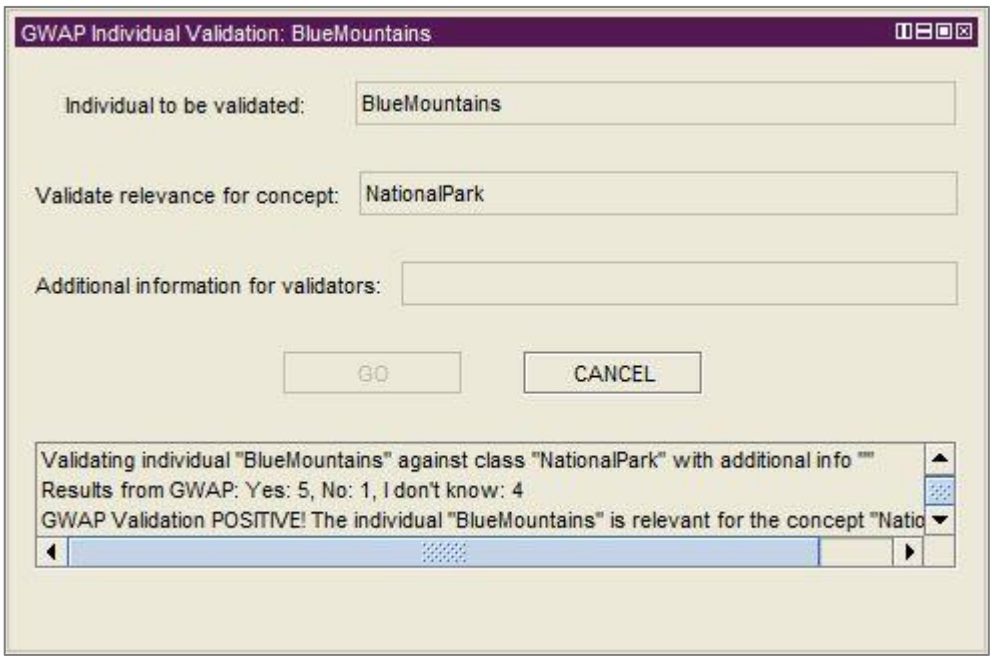


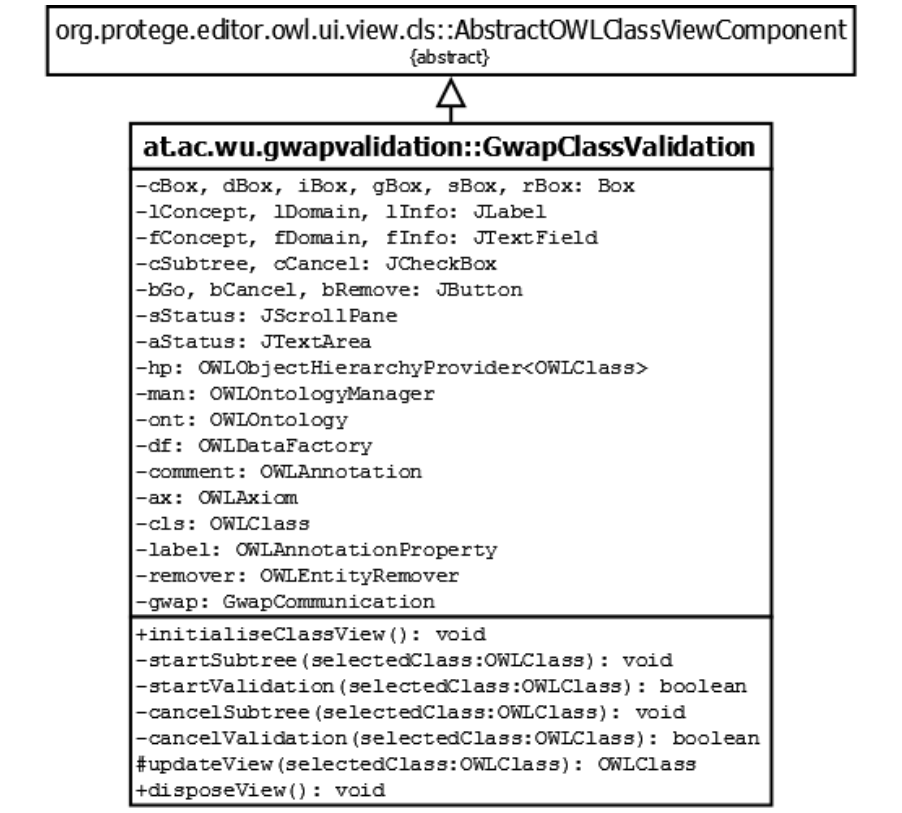
Abbildung 12: Protégé Plugin: GUI Individuen

Diese gleicht der Validierung von Konzepten sowohl von der GUI als auch vom Funktionsumfang zu einem großen Teil. Einzig die Validierung von Subtrees ist nicht implementiert, aus dem einfachen Grund, dass bei OWL Individuen keine Hierarchien definiert sind. Ein Individuum ist immer vom Typ einer oder mehrerer Konzepte/Klassen (sind es mehrere, werden alle Klassen durch Beistriche getrennt im zweiten Textfeld angeführt und gemeinsam evaluiert), es gibt jedoch keine Unter- oder Ober-Individuen. An Abbildung 12 erkennt man auch gut das Verhalten der GUI, wenn weniger Platz zur Verfügung steht. Die Text Area wird mit Scroll Bars ausgestattet und die Labels und Textfelder passen ihre Größe geringfügig an, dank des BoxLayouts jedoch immer optisch ansprechend.

3.2.2 Aufbau des Plugins

3.2.2.1 Klassenhierarchie

Anhand des Klassendiagramms (aufgeteilt auf Abbildung 13 und Abbildung 14) soll nun der Aufbau des Plugins erklärt werden.



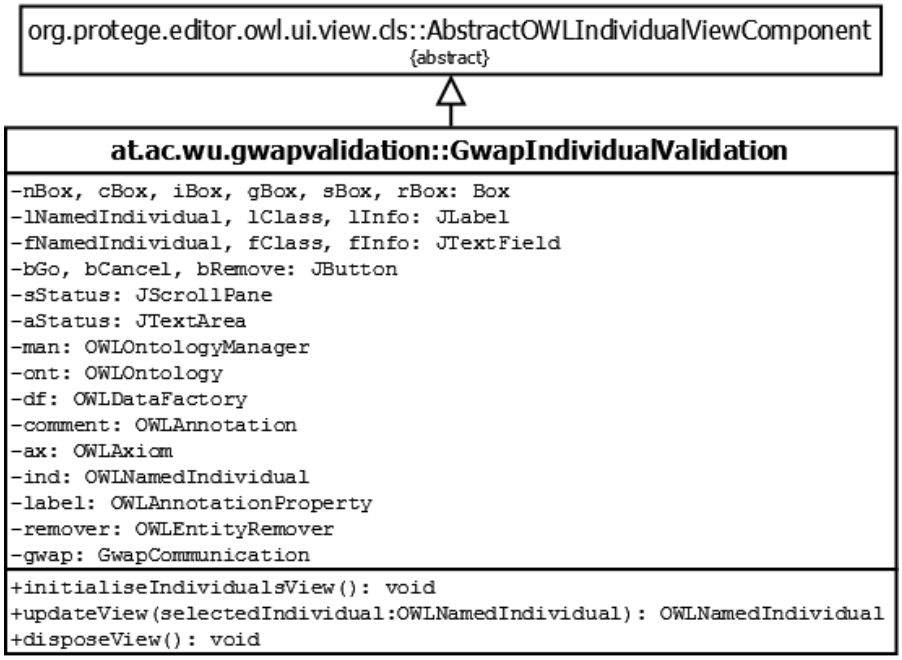


Abbildung 13: Protégé Plugin: Klassendiagramm 1

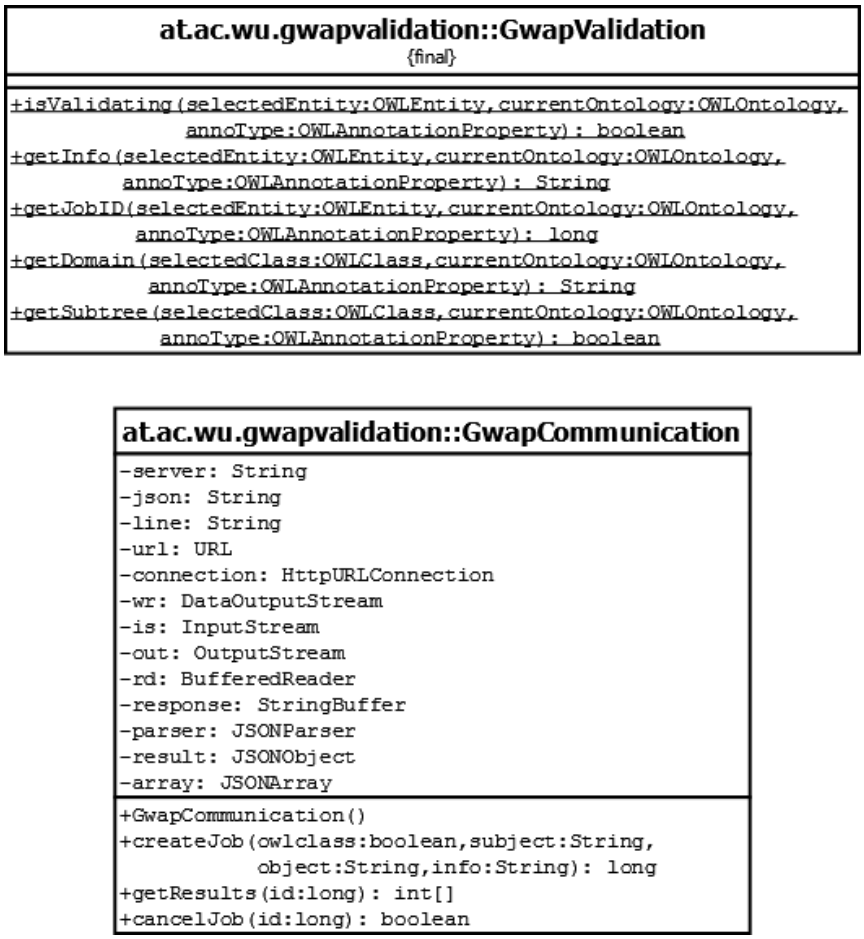


Abbildung 14: Protégé Plugin: Klassendiagramm 2

Wie im Klassendiagramm ersichtlich ist, besteht das Plugin aus insgesamt 4 Java Klassen, welche zu dem Paket *at.ac.wu.gwapvalidation* zusammengefasst wurden. Die Klasse *GwapClassValidation* wird von der Protégé Klasse *AbstractOWLClassViewComponent* abgeleitet und implementiert die Validierung von Konzepten. Analog dazu implementiert die Klasse *GwapIndividualValidation* die Validierung von Individuen und wird von der Klasse *AbstractOWLIndividualView* abgeleitet. Neben diesen beiden Hauptklassen gibt es noch die Klasse *GwapValidation*, welche einige Methoden beinhält, die sowohl von der Validierung der Konzepte als auch der Individuen benötigt werden, weshalb diese im Sinne der Modularität ausgelagert wurden. Im Normalfall wäre die optimalste Lösung für dieses Szenario, dass eine Oberklasse gebildet wird, von der die beiden anderen Klassen erben, und somit alle Methoden verfügbar haben. Da jedoch in Java keine Mehrfachvererbung möglich ist und beide Klassen von unterschiedlichen Protégé Klassen erben müssen, um die jeweilige Funktionalität der ViewComponent zu implementieren, musste eine andere Lösung gefun-

den werden. Interfaces, eine Möglichkeit für rudimentäre Mehrfachvererbung in Java, waren auch keine Möglichkeit, da in diesen nur die Signaturen vorgegeben und die Methoden nicht ausprogrammiert werden dürfen. Daher wurden die Methoden (ähnlich wie bei der Standard-Java-Klasse *Math*) in eine eigene finale Klasse ausgelagert und als statische Funktionen implementiert. Durch letzteres wird angezeigt, dass die Methoden nur mit den übergebenen Parametern arbeiten und keine Variablen der eigenen Klasse verwenden. Die vierte Klasse, *GwapCommunication*, dient den beiden Hauptklassen zur Verständigung mit dem externen GWAP, die komplette Kommunikation wurde zwecks Modularität in dieser Klasse programmiert. Nachfolgend werden die Komponenten der einzelnen Klassen näher beschrieben.

3.2.2.1.1 GwapClassValidation

Alle Attribute der Klasse *GwapClassValidation* angefangen von *cBox* bis hin zu *aStatus* sind GUI-Elemente und dienen daher zum Aufbau der grafischen Oberfläche bei der Validierung von Konzepten. Alle Attribute von *hp* bis *remover* sind Objekte spezieller Klassen für die Manipulation von OWL Elementen. Das Attribut *gwap* ist eine Instanz der Klasse GwapCommunication und dient der Validierung von Konzepten zum Datenaustausch mit dem GWAP.

Die Methoden *initialiseClassView*, *updateView* und *disposeView* müssen implementiert werden, dies wurde bereits in der Pilotstudie im Detail besprochen. Die Methoden *startValidation* und *cancelValidation* dienen dazu, eine Validierung für ein bestimmtes Konzept zu starten beziehungsweise zu stoppen. Mit den Methoden *startSubtree* und *cancelSubtree* kann eine Validierung für einen ganzen Teilbaum gestartet oder gestoppt werden.

3.2.2.1.2 GwapIndividualValidation

Die Klasse *GwapIndividualValidation* ähnelt von Aufbau her der Klasse *GwapClassValidation* sehr stark. Es gibt ein paar weniger Attribute, die teilweise auch anders benannt sind. Die vorhandenen Variablen erfüllen jedoch denselben Zweck wie in *GwapClassValidation*. Die Methoden beschränken sich auf die drei zwingend zu implementieren Funktionen. Da es keine Validierung von Teilbäumen bei den Individuen gibt, gibt es auch keine entsprechenden Metho-

den. Der vergleichbare Inhalt von *startValidation* und *cancelValidation* wurde hier direkt in den ActionListenern der jeweiligen Buttons implementiert.

3.2.2.1.3 GwapValidation

Die finale Klasse *GwapValidation* besitzt keine Attribute, jedoch fünf statische Methoden. Drei dieser Methoden werden von beiden Hauptklassen benötigt, zwei nur von *GwapClassValidation*. Zwecks Modularität und zukünftiger Erweiterbarkeit wurden jedoch alle Methoden nach GwapValidation ausgelagert.

Mit der Methode *isValidating* wird abgefragt, ob für ein bestimmtes Konzept oder Individuum bereits eine Validierung im Gange ist. Alle vier anderen Methoden dienen dazu, die Daten der Validierung (zusätzliche Information für die Validierenden; Job ID; Domäne, gegen die validiert wird; Information, ob ein ganzer Teilbaum validiert wird), die als RDFS Kommentar bei den entsprechenden OWL Klassen oder Individuen abgespeichert werden, abzufragen.

3.2.2.1.4 GwapCommunication

Alle Attribute der Klasse *GwapCommunication* angefangen von *server* bis hin zu *response* dienen zum Aufbau der Kommunikation und zum Datenaustausch mit dem GWAP über HTTP. Die Attribute *parser*, *result* und *array* sind JSON Objekte, die zur Verarbeitung der JSON Strings, die vom GWAP erhalten werden, benötigt werden.

Die Methode *GwapCommunication* ist der Konstruktor, der aktuell noch ohne Inhalt als Platzhalter fungiert. Mit den Methoden *createJob* und *cancelJob* wird ein Validierungsjob beim GWAP gestartet beziehungsweise gestoppt. Mit *getResults* werden die Ergebnisse einer Validierung vom GWAP abgerufen.

3.2.2.2 Benötigte Bibliotheken

In Abbildung 15 werden alle Libraries gelistet, die vom Plugin benötigt werden.

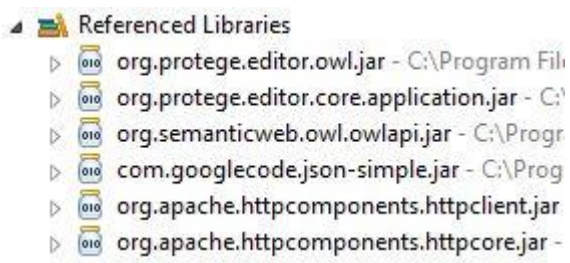


Abbildung 15: Protégé Plugin: Benötigte Bibliotheken

Die ersten drei Bibliotheken wurden bereits für den Prototyp benötigt. Die ersten beiden enthalten Kernklassen von Protégé selbst, wie zum Beispiel die ViewComponents. Die dritte Bibliothek enthält die Klassen der OWL API, welche ein eigenständiges Paket ist, jedoch auch als Plugin in Protégé eingebunden wird. Die enthaltenen Klassen sind zum Beispiel für die Manipulation einer Ontologie und ihren Axiomen notwendig. Die vierte Bibliothek ist json-simple, eine schlanke Implementation von JSON für Java. Die beiden letzten Bibliotheken beinhalten die HttpComponents von Apache für Java, welche im Plugin benötigt werden, um einen Datei-Upload per HTTP mit der POST-Methode durchzuführen. **Alle externen Libraries für JSON und HTTP müssen für die korrekte Einbindung als OSGi-Bundle vorliegen und mit dem GWAP Plugin gemeinsam im plugins-Ordner von Protégé abgelegt werden.** Für das korrekte Funktionieren des HTTP Clients wird noch eine weitere Bibliothek namens *commons-logging* benötigt. Diese muss nicht in Eclipse eingebunden werden, da das Plugin nicht direkt Klassen aus ihr verwendet. Jedoch verwenden die beiden anderen HTTP Client Bibliotheken Klassen aus *commons-logging*, daher muss diese auch im plugins-Ordner von Protégé abgelegt werden. Alle entsprechenden Download-Links für die Bibliotheken finden sich im Anhang.

3.2.3 Betrachtung und Analyse spezieller Komponenten

In diesem Abschnitt werden einzelne spezielle Komponenten beziehungsweise Konzepte des Plugins erläutert, die über die Verwendung der Java-Basis-Klassen hinausgehen und daher eine eigene Einarbeitung benötigt haben.

3.2.3.1 Manipulation von OWL Komponenten

Hier soll erläutert werden, wie die Bestandteile einer Ontologie aus einem Plugin heraus verändert werden können. Zu diesem Zweck werden die Klassen der OWL API verwendet. Für das Plugin war das Hinzufügen, das Entfernen und das Durchsuchen von RDFS Kommentaren notwendig (sowohl bei OWL Klassen und Individuen, als auch im Ontology Head), da in diesen die Daten für die Validierungen gespeichert werden. Die nachfolgend gezeigten Beispiele haben jedoch auch allgemeine Gültigkeit, da sie sich mit geringfügigen Änderungen auch auf die Manipulation sämtlicher anderer OWL Annotationen umlegen lassen.

3.2.3.1.1 Hinzufügen von Annotationen

Abbildung 16 zeigt das Hinzufügen einer Annotation im Kopf einer Ontologie.

```
// Syntax: "#gwap:domain=xyz;"
comment = df.getOWLAnnotation(df.getRDFSComment(),
    df.getOWLLiteral("#gwap:domain=" + fDomain.getText() + ";"));
man.applyChange(new AddOntologyAnnotation(ont, comment));
```

Abbildung 16: Hinzufügen einer Annotation zum Kopf einer Ontologie

Der erste Befehl erzeugt einen RDFS Kommentar als neue OWL Annotation. Hierfür wird auf die Instanz *df* der Klasse *OWLDataFactory* zurückgegriffen, mittels derer die Erzeugung von OWL Entitäten und Axiomen möglich ist. Mit dem zweiten Befehl wird die Annotation zur Ontologie hinzugefügt. Hierfür wird die Instanz *man* der Klasse *OWLOntologyManager* verwendet. Dies ist die Hauptklasse für die Verwaltung von Ontologien, mit ihr kann man Ontologien erzeugen, laden und verändern.

In Abbildung 17 wird das Hinzufügen einer Annotation zu einer OWL Klasse demonstriert.

```
// Syntax: "#gwap:domain=xyz;info=abc;subtree=true/false;jobid=123;"
comment = df.getOWLAnnotation(df.getRDFSComment(),
    df.getOWLLiteral("#gwap:domain=" + fDomain.getText() +
    ";info=" + fInfo.getText() + ";subtree=" + cSubtree.isSelected()
    + ";jobid=" + jobid + ";"));
ax = df.getOWLAnnotationAssertionAxiom(selectedClass.getIRI(), comment);
man.applyChange(new AddAxiom(ont, ax));
```

Abbildung 17: Hinzufügen einer Annotation zu einer OWL Klasse

Der erste Befehl erzeugt wiederum eine neue OWL Annotation. Mit dem zweiten Befehl wird ein neues Axiom erstellt, dieses bildet eine Verknüpfung zwischen zwei OWL Elementen (in diesem Fall OWL Klasse und Kommentar) ab. Die OWL Klasse wird anhand ihrer eindeutigen IRI identifiziert, analog könnte hier natürlich auch ein anderes Element (zum Beispiel OWL Individuum) angegeben werden. Der dritte Befehl fügt das Axiom zur Ontologie hinzu.

3.2.3.1.2 Durchsuchen und Entfernen von Annotationen

Abbildung 18 zeigt das Durchsuchen und Entfernen von Annotationen aus dem Kopf einer Ontologie.

```
for (OWLAnnotation anno : ont.getAnnotations()) {
    // if the comment starts with "#gwap:", it is relevant for the validation
    if (anno.getValue().toString().substring(1, 7).equals("#gwap:")) {
        man.applyChange(new RemoveOntologyAnnotation(ont, anno));
    }
}
```

Abbildung 18: Durchsuchen und Entfernen von Annotationen aus dem Kopf einer Ontologie

Das Durchsuchen von Annotationen lässt sich am besten mit der Kombination aus einer For-Each-Schleife und einer If-Abfrage lösen. Die Schleife geht alle Annotationen aus dem Kopf einer Ontologie durch, in der If-Abfrage wird geprüft, ob die aktuelle Annotation dem gesuchten Muster entspricht. Der Befehl innerhalb des If-Blocks zeigt das Entfernen der Annotation aus der Ontologie.

In Abbildung 19 wird das Durchsuchen und Entfernen von Annotationen einer OWL Klasse demonstriert.

```
for (OWLAnnotation anno : selectedClass.getAnnotations(ont, label)) {
    // if the comment starts with "#gwap:", it is relevant for the validation
    if (anno.getValue().toString().substring(1, 7).equals("#gwap:")) {
        ax = df.getOWLAnnotationAssertionAxiom(selectedClass.getIRI(), anno);
        man.applyChange(new RemoveAxiom(ont, ax));
    }
}
```

Abbildung 19: Durchsuchen und Entfernen von Annotationen einer OWL Klasse

Hier wird ebenfalls mit einer Kombination aus For-Each-Schleife und If-Abfrage gearbeitet. Die Schleife geht alle Annotationen einer OWL Klasse eines bestimmten Typs (Parameter *label*, kann zum Beispiel für RDFS Kommentar stehen) durch. In der If-Abfrage wird wiederum mit einem Muster verglichen. Analog zum Hinzufügen einer Annotation erzeugt der erste Befehl im If-Block ein Axiom, welches dann mit dem zweiten Befehl aus der Ontologie entfernt wird.

3.2.3.2 Validierung eines Teilbaums einer Ontologie

Zum Funktionsumfang des Plugins gehört die gemeinsame Validierung von Teilbäumen einer OWL Klassenhierarchie. Um dies zu ermöglichen, wird eine Rekursion eingesetzt, die es erlaubt, einen Baum unabhängig von seiner Struktur komplett durchzugehen. Zur Veranschaulichung wird in Abbildung 20 die *startSubtree*-Methode der Klasse *GwapClassValidation* dargestellt.

```
private void startSubtree(OWLClass selectedClass) {
    // start validation for current class
    startValidation(selectedClass);
    // call method recursively for every child class
    for (OWLClass sub : hp.getChildren(selectedClass)) {
        startSubtree(sub);
    }
}
```

Abbildung 20: Rekursion für die Validierung von Teilbäumen

Diese kurze Methode startet zuerst die Validierung für die aktuelle Klasse, danach wird für jede ihrer Kind-Klassen die Methode selbst erneut rekursiv aufgerufen. Um alle Kind-Klassen zu ermitteln wird die Instanz *hp* der Klasse *OWLObjectHierarchyProvider* verwendet, welche Methoden für die Abfrage des hierarchischen Aufbaus einer Ontologie bereitstellt. Die Kind-Klassen starten nun die Validierung für sich selbst, danach wird die Methode wiederum rekursiv für alle ihre Kind-Klassen aufgerufen, solange bis man am unteren Ende des Baums angekommen ist. Dadurch wird sichergestellt, dass ausgehend von der aktuellen Klasse unabhängig von der Baumstruktur für jede Unterklasse die Validierung gestartet wird. Auch die *cancelSubtree*-Methode basiert auf dem exakt gleichen Prinzip.

3.2.3.3 Kommunikation mit dem GWAP

Die Kommunikation mit dem GWAP wurde, wie bereits erwähnt, komplett in der Klasse *GwapCommunication* realisiert. Das GWAP läuft auf einem externen Webserver und bietet eine JSON API an, die durch den Aufruf entsprechender php-Dateien mittels HTTP-POST-Methode angesprochen wird. In allen drei Methoden der *GwapCommunication*-Klasse wird zuerst die Session zum Webserver aufgebaut. Danach werden (nur in der *createJob*-Methode) JSON Daten zum Server gesendet. Danach werden (wiederum in allen Methoden) JSON Daten vom Server empfangen. Diese werden in JSON Objekte geparst und abhängig vom Ergebnis weitere Schritte eingeleitet. In der *createJob*-Methode wird nach erfolgreicher Kommunikation der Metadaten auch noch eine csv-Datei mit den Spieldaten auf den Server hochgeladen. Bei all diesen Schritten muss natürlich auch ein Exception Handling implementiert werden, um mögliche Fehler in der Kommunikation abzufangen und entsprechend zu behandeln. Nachfolgend werden die einzelnen Schritte im Allgemeinen beschrieben.

3.2.3.3.1 Session Aufbau

In Abbildung 21 wird der Session Aufbau in der *createJob*-Methode dargestellt.

```
// connect to the webserver
url = new URL(server);
connection = (HttpURLConnection) url.openConnection();
// POST-method and JSON
connection.setRequestMethod("POST");
connection.setDoInput(true);
connection.setDoOutput(true);
connection.setRequestProperty("Content-Type", "application/json");
connection.setRequestProperty("Content-Length", "" +
    Integer.toString(json.length()));
```

Abbildung 21: GWAP Kommunikation: Session Aufbau

Die ersten beiden Befehle zeigen den Aufbau der HTTP Connection zur Server-URL. Mit den weiteren Befehlen werden die Parameter der Session konfiguriert. Es wird angegeben, dass die POST-Methode verwendet wird, dass sowohl gesendet als auch empfangen wird, dass der Inhalt der Kommunikation JSON ist und dass die Länge des Inhalts so lang wie der JSON String ist. Um diese Befehle und alle noch folgenden Befehle der Kommunikation wird ein globaler try/catch/finally-Block gebildet, der entsprechendes Exception Handling implementiert und zum Beispiel die Verbindung auch im Fehlerfall sauber terminiert.

3.2.3.3.2 JSON Daten senden

Abbildung 22 zeigt das Versenden des JSON Strings an den Webserver.

```
// send game metadata
wr = new DataOutputStream(connection.getOutputStream());
try {
    wr.writeBytes(json);
    wr.flush();
}
finally {
    wr.close();
}
```

Abbildung 22: GWAP Kommunikation: JSON Daten senden

Da die Metadaten zwar im JSON-Format, aber dennoch als String an den Server gesendet werden, funktioniert dies über die Erstellung und Verwendung eines *DataOutputStreams*, der an die HTTP Connection gekoppelt wird. Die Interpretation der JSON Daten erfolgt serverseitig.

3.2.3.3.3 JSON Daten empfangen

Abbildung 23 zeigt das Empfangen eines JSON Strings vom Webserver.

```
// get back response including job ID
is = connection.getInputStream();
rd = new BufferedReader(new InputStreamReader(is));
response = new StringBuffer();
try {
    while((line = rd.readLine()) != null) {
      response.append(line);
    }
}
finally {
    rd.close();
}
```

Abbildung 23: GWAP Kommunikation: JSON Daten empfangen

Hierfür wird ein *InputStreamReader* an die HTTP Connection gekoppelt, welcher wiederum einem *BufferedReader* zugewiesen wird. Über diesen können zeilenweise Daten gelesen werden, bis der Server keine Daten mehr sendet. Das Ergebnis ist ein *StringBuffer*, ein spezieller threadsicherer String. Auch hier werden die Daten wiederum nur empfangen, die Interpretation als JSON erfolgt erst im nächsten Schritt.

3.2.3.3.4 JSON Daten parsen

Abbildung 24 zeigt das Parsen der JSON Daten aus dem erhaltenen String.

```
// parse response string into JSON Object
result = (JSONObject) parser.parse(response.toString());

// if job creation was successful
if (result.get("success").equals("true")) {
```

Abbildung 24: GWAP Kommunikation: JSON Daten parsen 1

Das Objekt *parser* ist vom Typ *JSONParser*, mit der Methode *parse* erhält man ein Objekt der obersten Java-Klasse *Object*, dieses wird mittels Cast in ein *JSONObject* umgewandelt. Danach kann man mit der *get*-Methode des JSON-Objects den Wert eines beliebigen Namens auf der obersten Ebene abfragen, in diesem Fall ob der Inhalt des Namens beziehungsweise der Variable *success* dem Wert *true* entspricht. Will man tiefergehende Strukturen wie zum Beispiel die Elemente eines Arrays erreichen, muss wie in Abbildung 25 beim Parsen der Resultate des GWAPs vorgegangen werden.

```
// parse JSON data
result = (JSONObject) parser.parse(response.toString());
array  = (JSONArray) result.get("results");
result = (JSONObject) array.get(0);
```

Abbildung 25: GWAP Kommunikation: JSON Daten parsen 2

Hier arbeitet man sich Schritt für Schritt durch die JSON-Struktur. Nachdem die Antwort in ein *JSONObject* umgewandelt wurde, wird das Array namens *results* auf der obersten Ebene extrahiert. Nun kann wiederum mit der *get*-Methode auf die einzelnen Elemente des Arrays zugegriffen werden, indem der Index des gesuchten Elements als Parameter übergeben wird.

3.2.3.3.5 File-Upload

Abbildung 26 zeigt den Upload einer Datei an den Webserver des GWAPs. Der entscheidende Punkt ist hierbei die Klasse *MultipartEntity* aus der Apache HttpCore Bibliothek. Dieser wird die hochzuladende Datei zugewiesen, mittels *writeTo*-Methode kann die Datei dann über den *OutputStream* der HTTP Connection upgeloadet werden. Das Ganze muss natürlich wiederum mit entsprechendem Exception Handling versehen werden.

```
// file upload
FileBody fileBody = new FileBody(temp);
MultipartEntity multipartEntity =
    new MultipartEntity(HttpMultipartMode.STRICT);
multipartEntity.addPart("file", fileBody);

connection.setRequestProperty("Content-Type",
    multipartEntity.getContentType().getValue());

out = connection.getOutputStream();
try {
    multipartEntity.writeTo(out);
}
finally {
    out.close();
}
```

Abbildung 26: GWAP Kommunikation: File Upload

4. Ergebnisse

Das Ergebnis dieser Arbeit ist das fertig entwickelte Plugin für den Protégé Ontology Editor, mit dem Teile einer Ontologie zwecks Validierung an ein GWAP gesendet werden können. In der Endversion ist sowohl die Validierung von Konzepten (OWL Klassen) als auch Individuen (OWL Individuen) möglich, außerdem wurde die Validierung von Teilbäumen einer OWL Klassenhierarchie implementiert. Das externe GWAP ist zum Zeitpunkt des Abschlusses dieser Arbeit im Oktober 2013 noch nicht fertiggestellt. Jedoch ist die API vollständig definiert und vorhanden, sodass die komplette Kommunikation zwischen Plugin und GWAP bereits implementiert werden konnte. Dadurch sind keine Anpassungen mehr notwendig und das Plugin ist bereits jetzt im finalen Zustand.

5. Diskussion

Es soll nun betrachtet werden, wie sich das fertige Plugin zum jetzigen Zeitpunkt auf die Beantwortung der Forschungsfragen auswirkt.

Die spezielle Forschungsfrage S1 („Wie kann ein Plugin für den Protégé Ontology Editor zur Validierung von Ontologien mit Hilfe von Games with a purpose entwickelt werden?“) lässt sich durch die Ausführungen im praktischen Teil dieser Arbeit in Kapitel 3 beantworten. Es wurde gezeigt, dass es möglich ist, ein entsprechendes Plugin zu entwickeln und es wird im Detail beschrieben, wie dabei vorgegangen wurde.

Die beiden allgemeinen Forschungsfragen lassen sich zum jetzigen Zeitpunkt noch nicht gänzlich beantworten. Frage A1 („Sind Crowd-Sourcing Ansätze dafür geeignet, bei der Erstellung und Validierung von Ontologien hilfreich zu sein?“) ist laut den gesammelten Erfahrungen des Autors während der Entwicklung des Plugins vermutlich positiv zu beantworten, da durch das fertige Plugin nun die technischen Voraussetzungen dafür gegeben sind. Jedoch wird sich erst nach der Fertigstellung des GWAPs im produktiven Einsatz zeigen, ob das Crowd-Sourcing in diesem Fall wirklich sinnvolle Ergebnisse für die Validierung von Ontologien liefert. Frage A2 („Kann man teure Experten für die Erstellung und Validierung von Ontologien durch automatisierte Ansätze ersetzen, welche durch Crowd-Sourcing unterstützt werden, um eine bessere Skalierbarkeit zu

erreichen?“) muss getrennt betrachtet werden. Einerseits zeigt die Literatur, dass bei der Erstellung von Ontologien automatisierte Ansätze durchaus sinnvoll Anwendung finden können. Ob jedoch auch die Validierung automatisiert und durch Crowd-Sourcing unterstützt ablaufen kann und zumindest nahezu gleich gute Resultate wie bei der manuellen Validierung durch Experten erzielt werden, bleibt abzuwarten. Das Plugin muss erst über eine längere Zeit in der Praxis genutzt werden, um eine sichere Aussage darüber treffen zu können.

6. Zusammenfassung

Die Zielstellung dieser Arbeit war die Entwicklung eines Plugins für den Protégé Ontology Editor, mit dem Ontologien automatisiert mit der Hilfe eines GWAPs und Crowd-Sourcing validiert werden können. Im Rahmen dieser Arbeit wurden zuerst alle für das Projekt erforderlichen Konzepte und Technologien im Theorieteil erklärt. Danach wurde die konkrete Erstellung des Plugins beschrieben und dokumentiert. Da alle erforderlichen Funktionen des Plugins implementiert werden konnten, war die Entwicklung erfolgreich und die Zielstellung wurde erfüllt. Ob damit das übergeordnete Ziel, den Erstellungs- und Validierungsprozess von Ontologien zu verbessern und ressourcenschonender zu gestalten um die Skalierbarkeit des Semantic Web im Endeffekt zu verbessern, erreicht werden kann, wird sich erst im Laufe der Zeit zeigen.

Zur Weiterführung der Forschungstätigkeit ist eine gezielte Weiterentwicklung des Plugins sinnvoll. Zusätzliche Funktionen könnten die Validierung der taxonomischen Beziehungen, also den Zusammenhang zwischen Klassen und Subklassen, sowie die Validierung der Eigenschaften von Prädikaten betreffen. Sobald das externe GWAP fertiggestellt ist, soll eine Evaluierung der Crowd-Sourcing Ergebnisse im Vergleich zur Validierung durch Domain-Experten durchgeführt werden.

7. Anhang

Hier werden nützliche Links und Ressourcen zur Plugin Entwicklung angeführt.

Downloads für die Einrichtung der Entwicklungsumgebung:

- Java Development Kit:
  http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
- Protégé Ontology Editor:
  http://protege.stanford.edu/download/protege/4.3/installanywhere/Web_Installers/
- Graphviz:
  http://www.graphviz.org/Download..php
- Eclipse:
  http://www.eclipse.org/downloads/

Externe Bibliotheken:

- json-simple:
  http://code.google.com/p/json-simple/
- Apache HttpComponents (HttpClient und HttpCore):
  http://hc.apache.org/downloads.cgi
- Apache commons-logging:
  http://commons.apache.org/proper/commons-logging/download_logging.cgi

Java Dokumentationen (Javadoc):

- Java Allgemein:
  http://docs.oracle.com/javase/7/docs/api/
- Protégé:
  http://protege.stanford.edu/protege/4.3/docs/api/
- OWL API:
  http://owlapi.sourceforge.net/javadoc/
- json-simple:
  http://alex-public-doc.s3.amazonaws.com/json_simple-1.1/index.html
- Apache HttpClient:

http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/

- Apache HttpCore:
  http://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/

Ontologien:

- Travel Ontology:
  http://protege.cim3.net/file/pub/ontologies/travel/travel.owl
- Wine Ontology:
  http://www.w3.org/TR/owl-guide/wine.rdf

Protégé Entwickler Dokumentation (Plugin Entwicklung):

- Startseite (veraltet, Protégé 3):
  http://protege.stanford.edu/doc/dev.html
- Startseite (Protégé 4):
  http://protegewiki.stanford.edu/wiki/Protege4DevDocs
- Plugin Aufbau:
  http://protegewiki.stanford.edu/wiki/PluginAnatomy
- Plugin Typen:
  http://protegewiki.stanford.edu/wiki/PluginTypes
- Protégé APIs:
  http://protegewiki.stanford.edu/wiki/P4APIOverview
- Protégé Benutzeroberflächen-Komponenten:
  http://protegewiki.stanford.edu/wiki/P4UiComponentSummary

OWL API:

- Startseite:
  http://owlapi.sourceforge.net/index.html
- Dokumentation:
  http://owlapi.sourceforge.net/documentation.html
- Programmier-Beispiele:
  http://sourceforge.net/p/owlapi/code/ci/aef6981535f07a2d0d44c394b9f4d5415f36025a/tree/contract/src/test/java/org/coode/owlapi/examples/Examples.java

8. Literaturverzeichnis

[Bern01] Berners-Lee, Tim; Hendler, James; Lassila, Ora: The Semantic Web. In: Scientific American, May 2001, S. 29-37.

[Genn03] Gennari, John H. et al.: The evolution of Protégé: an environment for knowledge-based systems development. In: International Journal of Human-Computer Studies, Volume 58 Issue 1, January 2003, S. 89-123.

[Grub95] Gruber, Thomas R.: Toward principles for the design of ontologies used for knowledge sharing. In: International Journal of Human-Computer Studies, Volume 43, Issues 5–6, November 1995, S. 907–928.

[Hitz08] Hitzler, Pascal et al.: Semantic Web. 1. Aufl., Springer, Berlin 2008.

[Java13a] http://www.java.com/de/download/faq/whatis_java.xml, Abruf am 09.06.2013.

[Java13b] http://www.java.com/de/about/, Abruf am 09.06.2013.

[Json13a] http://json.org/json-de.html, Abruf am 14.06.2013

[Json13b] http://www.json.org/fatfree.html, Abruf am 14.06.2013

[KrHa11] Krüger, Guido; Hansen, Heiko: Handbuch der Java-Programmierung. 7. Aufl., Addison-Wesley, München 2011.

[MaSt01] Maedche, Alexander; Staab, Steffen: Ontology Learning for the Semantic Web. In: IEEE Intelligent Systems, Volume 16 Issue 2, March 2001, S. 72-79.

[Nurs09] Nurseitov, Nurzhan et al.: Comparison of JSON and XML Data Interchange Formats: A Case Study. In: Proceedings of the ISCA 22nd International Conference on Computer Applications in Industry and Engineering, CAINE 2009, November 4-6, 2009, S. 157-162.

[Orac13] http://www.oracle.com/us/technologies/java/features/index.html, Abruf am 09.06.2013.

[Prot13a] http://protege.stanford.edu/, Abruf am 14.07.2013

[Prot13b] http://protege.stanford.edu/overview/, Abruf am 14.07.2013

[Prot13c] http://protege.stanford.edu/doc/faq.html, Abruf am 14.07.2013

[Prot13d] http://protege.stanford.edu/aboutus/aboutus.html, Abruf am 14.07.2013

[RaSc09] Rafelsberger, Walter; Scharl, Arno: Games with a Purpose for Social Networking Platforms. In: HT '09 20th ACM Conference on Hypertext and Hypermedia. Torino, Italy, June 29 - July 01, 2009, S. 193-198.

[Sara12] Sarasua, Cristina; Simperl, Elena; Noy, Natalya F.: CROWDMAP: Crowdsourcing Ontology Alignment with Microtasks. In: ISWC'12 Proceedings of the 11th international conference on The Semantic Web - Volume Part I. Boston, MA, USA, November 11-15, 2012, S. 525-541.

[SiHe08] Siorpaes, Katharina; Hepp, Martin: OntoGame: Weaving the Semantic Web by Online Games. In: ESWC'08 Proceedings of the 5th European semantic web conference on The semantic web: research and applications. Tenerife, Canary Islands, Spain, June 1-5, 2008, S. 751-766.

[Shad06] Shadbolt, Nigel; Hall, Wendy; Berners-Lee, Tim: The Semantic Web Revisited. In: IEEE Intelligent Systems, Volume 21 Issue 3, May 2006, S. 96-101.

[Trav13] http://protege.cim3.net/file/pub/ontologies/travel/travel.owl, Abruf am 21.07.2013

[Ulle11] Ullenboom, Christian: Java ist auch eine Insel: Das umfassende Handbuch. 10. Aufl., Galileo Computing, Bonn 2011.

[Vahn06] von Ahn, Luis: Games with a Purpose. In: Computer, Volume 39 Issue 6, June 2006, S. 92-94.

[Weic11] Weichselbraun, Albert; Gindl, Stefan; Scharl, Arno: Using Games with a Purpose and Bootstrapping to Create Domain-Specific Sentiment Lexicons. In: CIKM '11 International Conference on Information and Knowledge Management. Glasgow, United Kingdom, October 24 - 28, 2011, S. 1053-1060.