Ruprecht-Karls-Universität Heidelberg
Neuphilologische Fakultät
Seminar für Computerlinguistik

Magisterarbeit

Semantische Agenten im Information Retrieval

Eine Studie über Semantic Web-Technologien

Wiebke Wagner

Betreuerin: Prof. Dr. Karin Haenelt
Zweitgutachter: Dr. Markus Demleitner

Heidelberg, 14.09.2007

Inhaltsverzeichnis

1 Einleitung

Das World Wide Web (WWW) mit geschätzten 16 Milliarden Seiten[1] stellt eine immense Quelle von Wissen dar. Dieser schier unerschöpfliche Fundus an Informationen ist aber nur in dem Maße wertvoll, wie die Informationen wieder aufgefunden werden können. Im gegenwärtigen Internet erinnert die Suche nach einer Information der sprichwörtlichen Suche einer Nadel im Heuhaufen. Suchmaschinen wie Google und AltaVista sind bei dem maschinellen Information Retrieval zwar nicht wegzudenkende Werkzeuge, allerdings ergeben sich einige ernsthafte Probleme im Umgang mit ihnen:

- Hohe Recall- und niedrige Precision-Werte.
  Selbst wenn die relevanten Seiten gefunden wurden, sind sie kaum nützlich, wenn daneben noch weitere zigtausend Treffer mit weniger oder gar nicht relevanten Dokumenten zurückgegeben werden.

- Niedriger oder kein Recall-Wert.
  Seltener aber oft genug passiert es, dass keine Treffer oder zumindest nicht die relevanten Seiten gefunden werden.

- Das Ergebnis ist stark von dem Vokabular abhängig, das in einer Suchanfrage verwendet wird. Oft gibt das gewählte Suchwort keine befriedigenden Ergebnisse, weil auf den relevanten Seiten eine andere Terminologie verwendet wird.

- Die Ergebnisse bestehen aus einzelnen Webseiten. Wenn man Informationen braucht, die auf mehreren Seiten verteilt sind, muss man mehrere Anfragen stellen, um die relevanten Dokumente zu erhalten. Anschließend müssen die partiellen Informationen aus den einzelnen Dokumenten manuell extrahiert und zusammengestellt werden.

- Genau genommen leisten die Suchmaschinen kein *Information Retrieval*, da sie nicht die Informationen finden, sondern lediglich den Ort, an dem die Informationen wahrscheinlich vorliegen. Den Prozess der Informationsextraktion aus den Seiten müssen menschliche Nutzer vollziehen.

[1] Vgl. WorldWideWebSiz.com: http://www.worldwidewebsize.com/

Aus diesen Gründen hat das Auffinden von relevanten Seiten immer mit Glück, Intuition und viel Erfahrung zu tun. Dass die Performanz von Suchmaschinen derart eingeschränkt ist, liegt daran, dass die gespeicherte Information überwiegend in natürlicher Sprache kodiert ist. Die Rezeption von Internetseiten ist damit stark auf den menschlichen Benutzer beschränkt. Maschinelle Systeme können die Seiten zwar lesen, sie können Texte in Sätze aufteilen, Wörter zählen, die Orthographie überprüfen etc. Der semantische Inhalt eines Textes ist für Programme aber nach wie vor sehr schwierig zu erfassen. Damit bleibt die Extraktion von Wissen und das Auffinden von Informationen problematisch.
Zur Überwindung des Problems hat das World Wide Web Consortium (W3C) – eine internationale Arbeitsgemeinschaft, die Standards und Richtlinien für das Internet entwickelt – die Semantic Web-Initiative ins Leben gerufen. Das Konzept des Semantic Web basiert auf einer Idee des W3C-Direktors und WWW-Begründers Tim Berners-Lee (2001). Dieser Idee zufolge sollen die Dokumente des WWW so aufbereitet werden, dass sie nicht nur von Menschen gelesen werden können, sondern dass auch Maschinen den Web-Inhalt zumindest so weit erfassen können, dass eine Automatisierung auf der Ebene der Semantik möglich wird.
Das Semantic Web ist kein neues Netz, sondern stellt eine Erweiterung des bestehenden Internet dar. Während man das gegenwärtige Web als dezentralisierte Platform für beliebige Präsentationen ansehen kann, die durch Verweise vernetzt sind, ist das Semantic Web eher mit einer dezentralisierten Wissensbasis zu vergleichen, einem Netz aus Inhaltsstrukturen. Dafür müssen die Inhaltsstrukuturen der Webseiten mit Hilfe von formalen Metasprachen erschlossen werden, die auf einem fest definierten Logiksystem basieren. Derart aufbereitete Webseiten besitzen eine maschinell verarbeitbare Semantik, die auch Inferenzprozesse zulässt.
Auf solchen Strukturen können semantische Software-Agenten operieren und Beziehungen zwischen Datenressourcen erkennen und verarbeiten. Wenn auch ein Agent die in der Metasprache kodierte Semantik nicht versteht wie ein Mensch, so ist er doch in der Lage, die Inhalte in einer Weise zu verarbeiten, die für den Menschen vernünftig erscheint und nützlich ist. Ein Agent kann im Semantic Web sehr effizient nach Informationen suchen, er kann Informationen sammeln, zusammenstellen und vergleichen. Er ist bei seinen Recherchen nicht nur auf statische Seiten angewiesen, sondern kann z.B. auch Web Services konsultieren. Vor allen Dingen ist er in der Lage, wirklich mit den Informationen umzugehen und nicht nur deren Webadressen zu liefern. Ein solcher Agent ist nicht nur für das Information Retrieval sehr nützlich. Er kann auch Flüge buchen, Arbeitsprozesse organisieren, Termine planen, in Online-Shops kaufen etc., um nur einige Anwendungen zu nennen.
Für die Umsetzung dieser vielversprechenden Idee hat das W3C bereits diverse Technologien entwickelt. Zudem gibt es inzwischen einige Semantic Web-Werkzeuge, die

die Bearbeitung dieser Technologien vereinfachen und anwenderfreundlich machen sollen. Trotzdem scheint es noch ein steiniger Weg zu sein, bis sich oben beschriebene Verhältnisse in vollem Umfang eingestellt haben.
In dieser Arbeit soll nun der Frage nachgegangen werden, wie praktikabel das Semantic Web für den Benutzer ist. Was sind die Probleme, die sich ihm stellen, wie detailiert muss er sich mit den Technologien, mit Logik und Semantik auskennen, um selbst Semantic Web-Seiten zu erstellen? Wie groß ist der Aufwand, ein neues Wissensgebiet semantisch zu erschließen, und wie groß ist der Aufwand, Seiten mit Metadaten zu versehen? Es wird untersucht, wie eine Webseite aussehen muss, so dass sie von einem Software-Agenten semantisch verarbeitet werden kann, und wie ein Web Service-Anbieter sein Angebot auch Agenten zur Verfügung stellen kann. Zudem wird untersucht, wie ein semantischer Agent ausgestattet sein muss und welche Fähigkeiten er braucht, um nützliche Ergebnisse im Information Retrieval zu erlangen.
Für die Beantwortung dieser Fragen wurde eine Studie durchgeführt, in der die relevantesten Semantic Web-Technologien ausprobiert wurden. Ziel war es, ein System zu erstellen, das Informationen mit Hilfe eines Agenten aus einem winzigen, selbst erstellten „Semantic Web“ extrahiert. Der Agent hat die Aufgabe, Hobby-Gärtnerei in Bezug auf die Aussaat von Gemüse zu beraten.
Die Arbeit ist so gegliedert, dass zunächst in Kapitel 2 ein Überblick über die Architektur und die Bestandteile des Semantic Web gegeben werden. Kapitel 3 erklärt die Planung, den Aufbau und das Ziel der Studie. Ferner wird die verwendete Datenbasis vorgestellt. In Kapitel 4 geht es um die Entwicklung und das Design der benötigten Ontologien. Es wird vorgeführt, wie Ontologien neu erstellt werden, wie bereits bestehende Quellen wiederverwertet werden können und welche Anpassungen dafür notwendig sind. In Kapitel 5 wird gezeigt, wie eine Internetseite aussehen muss, damit ein Agent Wissen daraus extrahieren kann. Hier geht es einerseits um die manuelle Annotation von Webseiten und andererseits um die automatische Annotation mit Hilfe eines XSL-Stylesheets. In Kapitel 6 werden einige Werkzeuge vorgestellt und deren Nutzen und Probleme gegenübergestellt. Kapitel 7 beschreibt die Eigenschaften und Fähigkeiten des Agenten. Ferner wird die Architektur und die Implementierung des Agenten vorgestellt. Im letzten Kapitel wird schließlich ein Fazit gezogen. Es wird überlegt, inwiefern das Semantic Web bereits existiert und mit welchen Problemen es zu kämpfen hat.

2 Semantic Web

Das Semantic Web besteht aus mehreren aufeinander aufbauenden Schichten, die in Abbildung 2.1 [2] dargestellt sind. Die unterschiedlichen Schichten sollen im folgenden beschrieben werden.

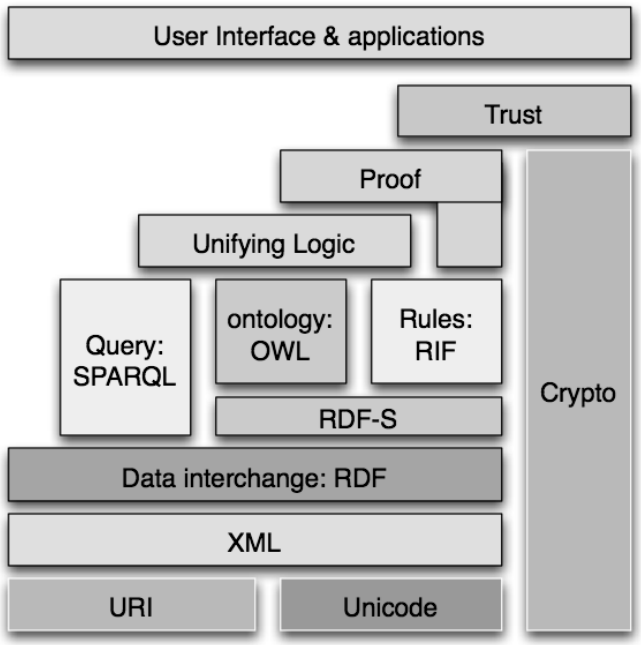


Abbildung 2.1: Das Schichten-Modell des Semantic Web

- XML bietet das Basisformat für die Dokumentstruktur, enthält aber noch keine spezielle Semantik.
- RDF ist ein Datenmodell, um einfache Aussagen über jegliche Objekte zu machen.
- RDF-Schema ermöglicht die Organisation von Objekten in Hierarchien. OWL, eine Ontologie-Sprache, ist eine Erweiterung von RDF-Schema und erlaubt die Repräsentation von komplexeren Relationen zwischen den Objekten. SPARQL dient dazu, die in OWL oder RDF kodierten Informationen wieder abzufragen. Mit RIF (Rule Interchange Format) soll der Ausdruck von Inferenzregeln

[2] Vgl. Berners-Lee 2006, http://www.w3.org/2006/Talks/0718-aaai-tbl/Overview.html#(14)

ermöglicht werden. RIF gehört insofern in die Nähe von Ontologien, als dass die Regeln Informationen der Ontologie kombinieren und dadurch neue Informationen gewinnen.

- Die Logik-Schicht reichert die Ontologie-Sprachen weiter zu einer turingkompletten Logiksprache an. Diese Sprache ist mächtig genug, um jegliche RDF-Anwendungen in Beziehung zueinander zu setzen. Sie definiert alles, was in den vorherigen Schichten noch nicht beschrieben werden konnte.

- *Proof* umfasst die Repräsentation von Beweisen in Web-Sprachen der unteren Schichten und die Bewertung von Beweisen. Dadurch wird ein Agent befähigt, die Richtigkeit seiner Informationen zu beweisen, indem er die Schlussfolgerungen aufführt, die ihn oder einen anderen Agenten zu seiner Antwort veranlasst haben.

- Die Spitze der Pyramide bildet die Anwendung von digitalen Unterschriften, die einem Agenten die Gewissheit verschaffen, dass eine Information aus einer zuverlässigen Quelle stammt. *Trust* ist ein kritisches und sehr wichtiges Kriterium: Nur wenn sich die Benutzer auf die Richtigkeit der Operationen und Informationen des Semantic Web verlassen können hat das Semantic Web Zukunft.

Die Technologien zur Realisierung des Semantic Web sind Empfehlungen und Arbeitsentwürfe des W3C. Die wichtigsten werden im Folgenden dargestellt.

2.1 XML, URI und XML-Namensräume

XML Die Extensible Markup Language (XML) ist eine Auszeichnungssprache des W3C, mit der hierarchisch strukturierte Daten dargestellt werden können. Sie ist insofern eine „Erweiterung“ zu HTML, als dass XML die Elementnamen frei definiert und unterschiedliche Daten wie Text, Graphik oder andere Medien beschreiben kann. Jeder Benutzer kann sein eigenes Dokumentformat definieren und in diesem Format schreiben, wobei die Definition in Schemasprachen wie DTD oder XML-Schema erfolgt. Im Gegensatz zu HTML ist die Datenstruktur strikt von ihrer Repräsentation getrennt. Dadurch lässt sich die gleiche Datengrundlage unterschiedlich darstellen, z.B. als Tabelle, als Text etc.
XML dient in erster Linie dem Austausch von Daten über das Internet und verfolgt daher das Ziel, sich auf möglichst einfache Weise im Internet nutzen zu lassen und ein breites Spektrum von Anwendungen zu unterstützen. XML soll formal und präzise und für Menschen lesbar sein. XML ist eine Sprache, mit der man leicht

Dokumente erstellen kann und für die man leicht Programme schreiben kann, die XML-Dokumente verarbeiten.
Es existieren inzwischen sehr viele formale Sprachen, die sich der Syntax von XML bedienen, so auch RDF, OWL und viele andere. XML ist ein wichtiger Faktor geworden, um eine Informationslandschaft zu schaffen, die sowohl für Maschinen als auch für Menschen verständlich ist.

URIs Eine Grundvoraussetzung für das Semantic Web ist die Möglichkeit, Dinge in der Welt eindeutig zu bestimmen. Erst wenn Dinge eindeutig identifiziert sind, können Aussagen über sie sicher zugeordnet werden. Dafür erweiterte man das Konzept der Universal Resource Locator (URL) dahingehend, dass eine Ressource nicht mehr unbedingt eine Internetadresse anzeigen muss, sondern auch zur Identifizierung einer beliebigen Ressource dienen kann. Man spricht hier von Universal Resource Identifier (URI). Ein URI ist lediglich ein Name für eine Ressource, die möglicherweise eine über das Internet erreichbare Pfadangabe ist, und die möglicherweise Information über die Ressource liefert. Beides muss aber nicht der Fall sein. URIs bilden eine Obermenge der URLs.

Namensräume Die Möglichkeit, Element- und Attributnamen in XML frei zu wählen, hat zur Folge, dass es leicht zu Kollisionen kommt. So ist es möglich, wenn nicht wahrscheinlich, dass es sowohl in einem Markup-Vokabular für Verlage als auch in einem Vokabular für Universitätsangestellte das Element *titel* gibt, das in beiden Vokabularen völlig unterschiedlich definiert ist. Um unterschiedliche Vokabulare miteinander mischen zu können, müssen die einzelnen Elemente und Attribute eindeutig einem Markup-Vokabular zugeordnet werden können. Dafür hat das W3C das Konzept der XML-Namensräume entworfen[3]. Ein XML-Namensraum ist eine Zusammenstellung von Namen, die durch eine URI identifiziert wird. Die Namensräume, die in einem XML-Dokument vorkommen, werden am Anfang des Dokuments definiert. Jedes Element und jedes Attribut des Dokuments bekommt ein Präfix, das mit Doppelpunkt abgesetzt wird. Dadurch werden die Sprachkonstrukte einem der angegebenen Namensräume zugewiesen. Wird kein Namensraum angegeben, gilt der lokale Default-Namensraum, der nur für das aktuelle Dokument gilt.

2.2 RDF

Mit XML lassen sich Daten flexibel in einer einheitlichen Sprache strukturieren; XML-Namensräume erlauben es, unterschiedliche XML-Vokabulare weltweit eindeutig zu definieren und mit XML-Schemata wird die Syntax der Vokabulare detailliert

[3] Vgl. Bray 2006

festgelegt. Der hierarchisch strukturierte XML-Baum bietet aber nicht die Möglichkeit, die Bedeutung von Daten maschinenlesbar zu definieren. Die Semantik bleibt ausschließlich vom Menschen interpretierbar.
Um dieses Problem zu lösen, wurden Methoden der Wissensrepräsentation aus der Künstlichen Intelligenz übernommen und für die praktische Nutzung im Web angepasst. Das Resource Description Framework (RDF) ist für diese Zwecke vom W3C 1999 bzw. 2004 als Empfehlung[4] herausgegeben worden und dient der Wissenskodierung. RDF ist genau genommen keine Sprache, sondern ein Datenmodell in Tripeln. Ein RDF-Tripel ist ein gerichteter Graph bestehend aus einem Knoten – der Ressource, einer Kante – der Eigenschaft und wieder einem Knoten, der den Wert beschreibt. Eine Ressource ist irgendein Objekt, das man beschreiben möchte: Autoren, Bücher, Orte, Zeiten, Gemüse etc. Jede Ressource hat einen eigenen URI, der sie eindeutig identifizierbar macht. Eigenschaften sind spezielle Ressourcen, die eine Relation zwischen zwei anderen Ressourcen ausdrücken, z.B. *ist geschrieben von, hat Titel, liegt neben, wird gesät* etc. Auch Eigenschaften haben einen eigenen URI. Der Wert ist entweder ein Literal oder wiederum eine Ressource, über die nach gleichem Schema eine Aussage, also ein neues Tripel, gemacht werden kann.
Abbildung 2.2 stellt einen kleinen RDF-Graphen dar, wobei Ressourcen als Ellipse, Relationen als Pfeil und der literale Wert als Rechteck abgebildet sind.

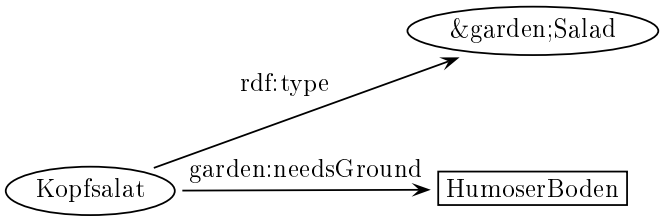


Abbildung 2.2: Beispiel eines RDF-Graphen

Die Repräsentation in Graphen ist zwar für Menschen eine übersichtliche Art der Darstellung, Computer arbeiten aber besser mit Darstellungen, die aus einer Folge von Zeichen bestehen, weshalb eine Serialisierung von RDF nötig ist. Eine von vielen Möglichkeiten ist die Serialisierung in XML, der Basis des Semantic Web. Die XML-Serialisierung des obigen Beispiels stellt sich dar wie in Listing 2.1. Weil XML-Code recht lang ist, gibt es einfachere Repräsentationen wie z.B. N3 oder Turtle, auf die hier aber nicht näher eingegangen werden soll.

```
1 <?xml version="1.0"?>
2
```

[4] Vgl. Resource Description Framework: http://www.w3.org/RDF/

```
3 <!DOCTYPE owl [
4 <!ENTITY garden "http://www.it-devel.de/semanticWeb/res/
5     ontologyo/01gartenOnto.owl#"
6 ]>
7 <rdf:RDF
8     xmlns:garden="http://www.it-devel.de/semanticWeb/res/
9         ontologyo/01gartenOnto.owl#"
10    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
11
12    <rdf:Description rdf:about="Kopfsalat">
13            <rdf:type rdf:resource="&garden;Salad"/>
14            <garden:needsGround>HumoserBoden</garden:needsGround>
15     </rdf:Description>
16 </rdf:RDF>
```

Listing 2.1: XML-Serialisierung eines RDF-Graphen

Ein RDF-Dokument wird in dem Element `rdf:RDF` repräsentiert (Zeile 7-16). Der Inhalt dieses Elements besteht einerseits aus der Definition der Namensräume (Zeilen 8-10) und aus einer beliebigen Anzahl von Aussagen über Ressourcen. Eine solche Aussage erfolgt in dem `rdf:Description`-Element, wobei die zu beschreibende Ressource über den URI in dem `rdf:about`-Attribut identifizierbar ist, hier `Kopfsalat` (Zeile 12). Wird eine Ressource neu gebildet, muss statt `rdf:about` das `rdf:ID`-Attribut verwendet werden. `rdf:type` (Zeile 13) ist eine RDF-Eigenschaft, die der Strukturierung dient. Sie weist eine Ressource ihrer Ontologie-Klasse zu – hier der Klasse `Salad` der Gartenontologie, die wiederum eine Ressource ist. Wenn die Eigenschaft eine Relation zu einem literalen Wert beschreibt, hat das Eigenschafts-Tag –hier `garden:needsGround` kein Attribut (Zeile 14). Der Wert wird statt dessen direkt in das Textfeld geschrieben.
Möchte man über mehrere Ressourcen als Ganzes eine Aussage machen, können Container-Elemente benutzt werden. Es gibt drei unterschiedliche Typen von Containern: `rdf:Bag` für einen ungeordneten Container, `rdf:Seq` für einen geordneten Container und `rdf:Alt` für eine Menge von Alternativen. Ein Container könnte in obigem Beispiel eingesetzt werden, wenn mehrere unterschiedliche Bodenqualitäten beschrieben werden sollen:

```
1     <rdf:Description rdf:about="Kopfsalat">
2         <garden:needsGround>
3             <rdf:Bag rdf:ID="Bodenqualitaet">
4                 <rdf:_1 rdf:resource="TiefgruendigerBoden"/>
5                 <rdf:_2 rdf:resource="DurchlaessigerBoden"/>
6                 <rdf:_3 rdf:resource="HumoserBoden"/>
```

```
7               </rdf:Bag>
8           </garden:needsGround>
9       </rdf:Description>
```

Listing 2.2: Beispiel eines Containers

Da die Reihenfolge der Bodeneigenschaften unerheblich ist und mit dem Container keine Alternativen geboten werden sollen, wird das `rdf:Bag`-Element benutzt. Der Container bildet dabei eine anonyme Ressource. Der Graph zu diesem Beispiel sieht aus wie in Abbildung 2.3

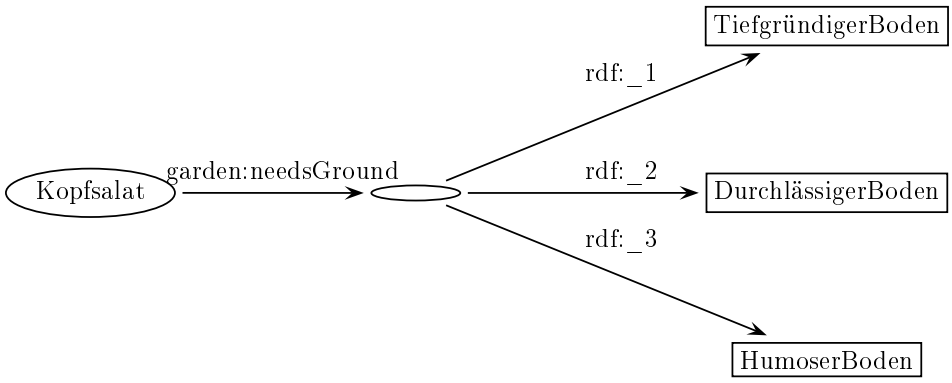


Abbildung 2.3: Beispiel eines RDF-Containers

Da RDF in Trippeln strukturiert ist, können nur binäre Prädikate nach dem Muster *prädikat(x,y)* dargestellt werden. Das Problem, mehr als zweistellige Prädikate auszudrücken, kann mit Hilfe einer anonymen Ressource innerhalb eines Containers umgangen werden.
Mit Containern kann nicht definiert werden, dass neben den aufgeführten Container-Elementen keine weiteren Elemente mehr bestehen. Um gesschlossene Gruppen auszudrücken, gibt es die RDF-Collections. RDF-Collections entsprechen einer verketteten Liste. Die Eigenschaften `rdf:first` und `rdf:rest` verweisen solange auf das nächste Listen-Element, bis die Ressource `rdf:nil` erreicht ist. Das Attribut `rdf:parseType=Collection` ermöglicht eine vereinfachte Schreibweise der Listenstruktur. Es kann mit einer beliebigen Eigenschaft verwendet werden, wobei die Listenelemente in einem `Description`-Element angegeben werden. Listing 2.3 zeigt den Quellcode einer RDF-Collection:

```
1   <rdf:Description rdf:about="Tomato">
2       <garden:hasColour rdf:parseType="Collection">
3           <rdf:Description rdf:about="red"/>
4           <rdf:Description rdf:about="yellow"/>
5       </garden:hasColour>
```

```
6       </rdf:Description>
```

Listing 2.3: Beispiel einer RDF-Collection

Diese Aussage gibt an, dass Tomaten nur rot oder gelb sein können, nicht aber blau oder von irgendeiner anderen Farbe.
RDF enthält einen ausgesprochen mächtigen Mechanismus, der als *Reifikation* bezeichnet wird. In RDF ist es möglich, Aussagen über Aussagen zu machen. Dabei wird ein ganzes Tripel, also eine Aussage, wie eine Ressource behandelt, über die man wiederum etwas aussagen kann. Reifikation ist beispielsweise dann nützlich, wenn man mit unterschiedlichen Datenquellen arbeitet und ausdrücken möchte, dass *A* aus einer unzuverlässigen Quelle stammt. Reifikation macht logisches Schließen aber unentscheidbar, weil Paradoxien wie: *Diese Aussage ist falsch.* formulierbar werden. Antoniou et al. (2004, S.69) kritisieren zu Recht, dass ein so ausdrucksstarker Mechanismus in einer einfachen Sprache wie RDF nicht richtig aufgehoben ist, sondern auf eine höhere Stufe des Semantic Web-Schichtenmodells gehört.
Eine große Herausforderung des Semantic Web stellt das Problem dar, die vielen bestehenden Seiten des WWW mit RDF zu annotieren. Es ergibt sich von selbst, dass der Aufwand, jede Seite von Hand zu bearbeiten, nicht realistisch ist. Daher gibt es verschiedene Forschungsansätze, wie eine automatische oder semi-automatische Annotation umgesetzt werden könnte. Diese Verfahren basieren auf verschiedenen NLP-Verfahren[5], wurden bei dieser Studie aber nicht mitberücksichtigt.

2.3 Ontologien im Semantic Web

RDF ist eine Sprache, mit der man mit seinem eigenen Vokabular Ressourcen beschreiben kann. Die Semantik ist aber noch nicht formal definiert. Die Fixierung des Vokabulars erfolgt erst durch Ontologien, denen die RDF-Ressourcen zugewiesen werden können.
Eine Ontologie ist nach Antoniou (2005, S. 1) *a formal specification of a conceptualisation.* Mit anderen Worten ist eine Ontologie eine abstrakte und vereinfachte Ansicht auf die Welt, die repräsentiert werden soll – ein Modell also, das mit Hilfe von Ontologiesprachen zum Ausdruck gebracht wird. Neben einer wohl definierten Syntax wie XML muss die Ontologiesprache mit formaler Semantik ausgestattet sein, die die Bedeutung von Wissen so beschreibt, dass sie keinerlei unterschiedliche Interpretation zulässt. Dafür werden verschiedene Logiksysteme verwendet. Durch die unumstößliche Eindeutigkeit der Ontologie wird die Kommunikation zwischen menschlichen und/oder maschinellen Akteuren stark verbessert und der Austausch von Wissen erleichtert.

[5] Vgl. Granitzer, 2006

Das Semantic Web soll nicht eine große, alles enthaltende Ontologie zur Verfügung stellen, die von irgendeinem Organ zentral gesteuert werden müsste und in ihrem Umfang kaum zu handhaben wäre. Das Ziel ist vielmehr, dass eine große Anzahl von domänenspezifischen Ontologien entstehen, die der Benutzer in seinen Anwendungen je nach Bedarf zusammenstellt.
Um eine Wissensdomäne formal zu beschreiben, müssen einerseits die individuellen Objekte einer Domäne, die Ressourcen, spezifiziert werden und andererseits die Klassen, denen diese Objekte zugeordnet werden können. Eine Klasse wird hier als Menge mit Elementen verstanden. Die Individuen dieser Klasse werden auch als Instanzen bezeichnet. Um die Instanzen der einzelnen Klassen in ein Verhältnis zueinander zu setzen, werden Eigenschaften benötigt. Eigenschaften einer Menge von Instanzen können mit Beschränkungen belegt werden. Beschränkungen sind z.B. notwendig, um unsinnige Tripel wie *(Fruchtbarer Boden) (wird gesät) (Ende Februar)* zu vermeiden. Da in der Garten-Domäne nur Pflanzen gesät werden können, ist eine Beschränkung der Instanzen, auf die die Eigenschaft angewendet werden kann, notwendig. Man spricht hier von einer Beschränkung des Definitionsbereichs. Auch für den Wertebereich einer Eigenschaft bedarf es häufig einer Beschränkung, um Aussagen wie *(Kopfsalat) (wird gesät) (Tomate)* zu vermeiden. Es gibt weitere Beschränkungen, von denen später noch die Rede sein wird.
Klassen und Eigenschaften mit ihren Beschränkungen sind die Bestandteile einer Ontologie. Eine Ontologie, die mit Instanzen bestückt ist, wird meist als Wissensbasis bezeichnet[6].

2.3.1 Reasoning mit Ontologien

Nach May (2006, S.485) bedeutet Reasoning: *die Semantik [...] von Dingen zu nutzen, und sich dabei auf einer festen Basis, einer Logik zu bewegen. Somit ist Reasoning im Wesentlichen gesunder Menschenverstand, Verstehen und Nutzen der Bedeutung von Fakten.* Die Formalisierung der Semantik innerhalb einer Ontologie erlaubt es also, zusätzliche Aussagen aus der Ontologie zu erschließen, wie z.B.[7]:

- Klassenzugehörigkeit: Ist A eine Instanz der Klasse C und ist C eine Unterklasse der Klasse D, dann kann gefolgert werden, dass A auch eine Instanz von D ist.

- Äquivalenz von Klassen: Wenn Klasse C äquivalent zu Klasse D und D äquivalent zu Klasse E ist, dann sind auch C und E äquivalent.

[6] Vgl. Noy et al. (2001), S. 3
[7] Vgl. Antoniou, 2004, S. 110

- Konsistenz: Sind die Klassen C und D disjunkt zueinander und ist die Klasse E die Unterklasse von sowohl C als auch D, dann kann die Klasse E in keinem Fall Instanzen haben. Wenn Klassen immer leer sein müssen, spricht man von Inkonsistenz.

- Klassifikation: Man kann alle Unterklassen-Verhältnisse von Klassen erschließen. Wenn z.B. C eine Unterklasse von D und D eine Unterklasse von E ist, so ist E notwendigerweise auch eine Unterklasse von C.

Ableitungen wie die oben beschriebenen können automatisch von Programmen vollzogen werden. Besonders bei der Entwicklung von großen Ontologien mit mehreren Entwicklern und mehreren eingebundenen Ontologien sind solche Überprüfungen für das Debugging unverzichtbar.
Ontologiesprachen bewegen sich immer in dem Spannungsfeld von Ausdrucksstärke und Reasoning-Tauglichkeit. Je mächtiger eine Sprache ist, um so schlechter wird ihre Berechenbarkeit und um so schlechter lassen sich logische Schlussfolgerungen ziehen. Reasoning ist so lange möglich, wie eine Sprache entscheidbar ist, d.h. dass es für die beschriebene Semantik ein System geben muss, das nur gültige Schlüsse zieht und das für jede Aussage feststellen kann, ob sie wahr oder falsch ist.

2.3.2 Ontologiesprachen des Semantic Web

Da die Anforderungen an eine Ontologie je nach Anwendung sehr unterschiedlich sind, hat das W3C ein Schichtenmodell von Ontologiesprachen mit unterschiedlicher Ausdrucksstärke entwickelt. Alle Sprachen basieren auf RDF, wobei es für RDF-Schema (RDF-S) und die Web Ontology Language (OWL) jeweils einen zusätzlichen Namensraum gibt. Jedes RDF-S- und OWL-Dokument ist also auch ein RDF-Dokument.
RDF/RDF-S und OWL unterliegen unterschiedlichen Logikmodellen. Während Erstere mit Frame-Logic bearbeitet werden, basiert OWL auf Description Logic (DL), die eine Untermenge der Prädikatenlogik ist. Im Gegensatz zu ihr ist sie aber entscheidbar. Eine Wissensbasis in DL unterteilt sich in die *TBox*, die terminologogical Box, die das intensionale Wissen der Wissensbasis enthält und die *ABox*, assertional Box, die die einzelnen Instanzen beschreibt und somit das extensionale Wissen enthält. Instanzen und Klassen sind also strikt getrennt. Die in der ABox gegebenen Fakten bedeuten im DL-Reasoning das „sichere“ Wissen über Instanzen aber nicht unbedingt das vollständige Wissen. Wenn für eine Buschbohne gesagt wird, dass sie grün ist, so würde ein DL-Reasoner die Frage: *ist eine Buschbohne blau* nicht mit nein beantworten, weil nicht ausdrücklich gesagt ist, dass Buschbohnen ausschließlich grün sind. Dies bezeichnet man als *Open-World-Assumption*.

ALC ist die minimale DL-Sprache, die bereits Disjunktion und Negation voll unterstützt. Sie kann semantisch beliebig erweitert werden. Die Grundlage für OWL bildet die Description Logic ALC_{R+}, die zusätzlich transitive Rollen enthält und auch als S bezeichnet wird. Ferner dürfen in OWL Rollen hierarchisch geordnet sein (H) und können inverse Rollen definiert werden (I). Auch qualifizierte Kardinalität (Q) angewendet auf nicht-transitive Rollen ist erlaubt. Zusammen ergibt sich $SHIQ$, was noch eine entscheidbare Sprache ist[8].

RDF-Schema

Das grundlegende Element eines RDF-Schemas ist die Klasse, der Instanzen zugewiesen werden können. Klassen können mit der Eigenschaft `rdfs:subClassOF` in eine hierarchische Struktur gebracht werden, wobei die Unterklasse die Eigenschaften der Oberklasse erbt, zusätzlich aber noch weitere Eigenschaften enthält. Unterklassen sind also eine Spezialisierung der Oberklasse. Ebenso wie Klassen können auch Eigenschaften hierarchisch angeordnet werden.
Die einzigen Beschränkungen auf Eigenschaften, die mit RDF-S möglich sind, beziehen sich auf den Definitions- und Wertebereich. Damit ist RDF-S zu nicht viel mehr in der Lage, als eine taxonomische Strukturierung zu repräsentieren. RDF-S enthält aber zumindest soviel ontologisches Wissen, um ein Vokabular in Bezug auf die Ober- und Unterbegriffe zu fixieren. Diese einfachen Modellierungskonzepte erlauben bereits einfache Inferenzen. Wenn z.B. eine Klasse C der Definitionsbereich einer Eigenschaft P ist und eine Instanz A die Eigenschaft P hat, so folgt daraus, dass A Instanz der Klasse C sein muss.
Auch wenn RDF-S eine sehr primitive Ontologiesprache ist, enthält sie wie auch RDF einige sehr mächtige Mechanismen. In RDF-S gibt es keine strikte Trennung zwischen Instanz und Klasse, d.h. dass auch Klassen Instanzen sein können, über die man Aussagen machen kann. Dieser Mechanismus dient der Modellierung auf Metadaten-Ebene. Er hat zur Folge, dass die Sprache nicht mehr entscheidbar ist. Z.B. sind dadurch Paradoxien formulierbar wie: *die Menge aller Klassen, die sich nicht selbst enthalten.*

OWL Lite

OWL Lite baut zwar syntaktisch auf RDF-S auf, schränkt aber die Semantik so weit ein, dass sie entscheidbar bleibt. Klassen, Instanzen und Eigenschaften sind disjunkt, damit oben beschriebene Paradoxien nicht zustande kommen können. OWL Lite enthält die folgenden Konstrukte[9]:

[8] Vgl. May 2006, S. 498
[9] Vgl. May, 2006, 498

- atomare Klassen und Klasseneinschränkungen über Restriktionen von Eigenschaften

- Gleichheit und Verschiedenheit von Klassen (`owl:sameAs, owl:differentFrom`)

- Existenzquantor und Allquantor (`owl:someValuesFrom, owl:allValuesFrom`)
  Eine Beschränkung mit dem Existenz-Quantor spezifiziert für eine Menge von Individuen die Existenz einer Eigenschaft zu einem anderen Individuum einer Klasse, z.B. $\exists$ *hasChild Male* schließt alle Instanzen einer Klasse ein, die mindestens ein männliches Kind haben. Durch die Beschränkung wird, wie bei allen anderen Beschränkungen auch, eine anonyme Klasse gebildet, die diejenigen Instanzen enthält, die die Bedingung erfüllen. In hiesigem Beispiel würde die anonyme Klasse alle Instanzen enthalten, die ein männliches Kind haben. Mit dem All-Quantor kann eine Eigenschaft insofern beschränkt werden, als das alle Werte dieser Eigenschaft von einer bestimmten Klasse kommen müssen. Die Beschränkung $\forall$ *hasChild Male* schließt die Instanzen einer Klasse ein, die nur männliche Kinder haben. Beschränkungen mit dem Allquantor spezifizieren nicht die Existenz einer Relation. Sie besagen lediglich: wenn die beschränkte Relation besteht, dann muss sie zu einem Individuum der bestimmten Klasse gehen. Auch Individuen ohne die beschränkte Eigenschaft sind Instanzen der anonymen Klasse. Instanzen ohne *hasChild*-Eigenschaft haben folglich keine *hasChild*-Eigenschaft, die nicht mit einem Individuum der Klasse *Male* verbunden ist. Damit ist die Beschränkung erfüllt.

- Mindest- und Maximalkardinalität mit dem Wert 1 (`owl:cardinality, owl:minCardinality, owl:maxCardinality`)
  Mit Kardinalität wird beschränkt, wie oft eine Eigenschaft eines Individuums vorliegen muss. Bei einer Mutter müsste die Relation *hasChild* auf eine Mindestkardinalität von $\geq$1 beschränkt sein, da die Mutter mindestens ein Kind haben muss; möglicherweise sind es aber auch mehr.

- Symmetrie, Inverse und Transitivität von Eigenschaften
  Symmetrie ist dann gegeben, wenn ein Individuum A eine Relation zu Individuum B hat und B die gleiche Relation zu A hat, z.B. das Verhältnis von Geschwistern: *Anna hasSibling Peter / Peter hasSibling Anna.*
  Wenn eine Eigenschaft Individuum A und B miteinander verbindet, dann folgt daraus die korrespondierende inverse Eigenschaft, die Individuum B mit A verbindet, z.B. das Eltern-Kind-Verhältnis: *Anna hasChild Peter / Peter hasParent Anna*, wobei *hasChild* und *hasParent* zueinander inverse Eigenschaften sind.

Eine Eigenschaft ist transitiv, wenn sie ein Individuum A mit einem Individuum B und B mit C verbindet. Daraus folgt, dass die Eigenschaft auch für A nach C gilt, z.B. das Verhältnis von Vorfahren: Aus *Peter hasAncestor Anna* / *Anna has Ancestor Paul* folgt *Peter hasAncestor Paul*. Transitivität darf nicht bei komplexen Eigenschaften deklariert werden, d.h. die Eigenschaft darf nicht eine Inverse oder Subeigenschaft einer komplexen Eigenschaft sein und sie darf keine Kardinalitätseinschränkung haben.

- Benutzung von XML-Schema-Datentypen
  In OWL gibt es zwei Arten von Eigenschaften: Objekt-Eigenschaften, die Objekte mit anderen Objekten in Beziehung setzen und Datentyp-Eigenschaften, die einem Objekt den Wert eines bestimmten Datentyps zuordnet, z.B. Alter: *Peter hasAge 99*.

OWL Lite hat damit eine Ausdruckskraft, die der Beschreibungslogik *SHIF(D)* entspricht. F drückt die Einschränkung der Kardinalität aus, die nur die Werte 0, 1 und *beliebig* erlaubt. D steht für konkrete Datentypen.

OWL-DL

Bei OWL-DL kommen noch einige Sprachkonstrukte hinzu, die die Sprache mit größerer Ausdrucksstärke ausstatten:

- Disjunktion von Klassen, d.h. die Klassen dürfen keine gemeinsamen Instanzen haben (`owl:disjointWith`)

- Definition von komplexen Klassen mittels Boolschen Operatoren (Vereinigung, Schnittmenge und Komplement)
  Wenn zwei oder mehr Klassen miteinander vereinigt werden, entsteht eine anonyme Klasse. Die Vereinigung der Klassen $Man \cup Woman$ bildet z.B. eine anonyme Klasse, deren Instanzen entweder zur Klasse *Man* oder zur Klasse *Woman* oder zu beiden Klassen gehören.
  Ähnlich verhält es sich mit Schnittmengen. Die Schnittmenge der Klassen $Human \cap Male$ bildet eine anonyme Klasse, deren Instanzen sowohl zur Klasse *Human* als auch *Male* gehören. Die anonyme Klasse ist sowohl eine Unterklasse von *Human* als auch von *Male*
  Eine Komplementklasse enthält alle Instanzen, die nicht in der Klasse enthalten sind, zu der sie das Komplement ist. Die Klasse *Male* ist z.B. das Komplement zu Klasse $\neg Male$ bzw. *Female*. Diese Aussage könnte auch mit `owl:disjointWith` ausgedrückt werden.
  Die Boolschen Ausdrücke können sowohl für Äquivalenz-Relationen als auch für die Relation von Unterklassen verwendet werden.

- Beliebige Mindest- und Maximalkardinalität
  Im Gegensatz zu OWL Lite ist die Kardinalität nicht auf die Werte 0 und 1 beschränkt.

- Klassendefinition durch Aufzählung der Instanzen (`owl:oneOf`)
  Eine Klasse *DaysOfWeek* könnte z.B. mit *Montag, Dienstag, ..., Sonntag* definiert werden.

- Werteeinschränkungen von Eigenschaften (`owl:hasValue`)
  Diese Beschränkung beschreibt eine anonyme Klasse, deren Individuen mit einem bestimmten anderen Individuum verbunden werden über eine gegebene Relation. Die Beschränkungen mit Quantoren sind im Gegensatz dazu mit irgendeinem Individuum einer bestimmten Klasse verbunden.

OWL-DL entspricht der Beschreibungslogik *SHOIQ(D)*. Im Gegensatz zu OWL Lite ist F mit Q ersetzt, wodurch die Kardinalitätseinschränkung aufgehoben wird. Ferner kommt O dazu. O steht in DL für *Nominals* und bezieht sich auf die Erweiterung von `owl:hasValue` und `owl:oneOf`.

OWL Full

Bei OWL Full kommen keine neuen Konstrukte mehr hinzu. Allerdings sind alle syntaktischen Kombinationen von RDF-S und OWL-DL erlaubt. OWL Full bietet also die Möglichkeit zur Meta-Modellierung. Es können sogar Sprachkonstrukte auf sich selber angewendet werden. Z.B. ist es erlaubt eine Kardinalitätsbeschränkung auf die Eigenschaft `rdfs:subClassOf` anzuwenden, was in OWL Lite und OWL-DL nicht möglich ist, weil Sprachkonstrukte dort nicht als semantische Objekte behandelt werden. Deshalb sind die beiden anderen OWL-Subsprachen keine Unterklasse von OWL Full. Es gilt:

$$OWL\ Lite \subset OWL - DL$$

$$RDF - S \subset OWL\ Full.$$

Eine konsequente Abwärtskompatibilität der Semantic Web-Ontologiesprachen ist nicht gegeben.

3 Eigenes Versuchsfeld

3.1 Ziel

Mit der Studie soll die Programmierung des Semantic Web erprobt und die vielen ineinander greifenden Technologien und ihr Zusammenwirken erklärt werden. Dafür werden Seiten aus dem WWW mit RDF annotiert, Ontologien entwickelt, die das Vokabuar für die annotierten Webseiten definieren und schließlich ein Agent implementiert, der mit Hilfe der RDF-Seiten und der Ontologien bestimmte Informationen aus den Seiten extrahieren kann.
Die Anwendung hat das Ziel, dass der Agent auf eine Anfrage eigenständig die geforderten Informationen zusammensucht. Der Agent muss dafür verschiedene Webseiten und einen Web Service nutzen und teilweise mit Hilfe der Ontologie die Antwort selbst erschließen. Dem Benutzer wird damit erspart, sich sämtliche Informationen in einer Suchmaschine selbst zusammenzusuchen.

3.2 Wissensgebiet

Zunächst musste ein geeignetes Testfeld gefunden werden, das den Nutzen der Semantic Web-Technologie möglichst gut illustriert. Die Semantic Web-Anwendung sollte ein möglichst realistisches Szenario wiedergeben und nützliche Antworten liefern. Die Informationen sollten nicht in einer einzigen Ressource zu finden sein, sondern mit Suchmaschinen nur mühsam über verschiedene Anfragen erhältlich sein. Die Information sollte ferner möglichst unterschiedlich auf den Seiten vorliegen, z.B. in Tabellen, in unstrukturiertem Text etc. Der Agent sollte Links verfolgen, die ihn auf relevante Seiten verweisen. Ferner sollte er eigenständig Web Services als Informationsquelle nutzen.
Diese Kriterien lassen sich beispielsweise im Wissensgebiet der Hobby-Gärtnerei finden. Das hier implementierte System – der Vegi-Agent – gibt Auskunft über die Anzucht von Gemüse. Er soll für jedes Gemüse zu folgenden Punkten Information aufsuchen:

- Bodenbeschaffenheit, die das Gemüse braucht, z.B. fruchtbarer Boden, lockerer Boden etc.

- der für das Gemüse günstigste Standort, z.B. sonniger Standort, windgeschützter Standort etc.

- Nachbarschaftsverhältnisse
  In Mischkulturen wird darauf geachtet, dass sich benachbarte Pflanzen in ihrem Gedeihen fördern. So gibt es Pflanzen, die sich gegenseitig „gute“ oder auch „schlechte“ Nachbarn sind. Die guten Nachbarn einer gegebenen Gemüsesorte soll der Agent ermitteln.

- Saatzeit
  Der Agent gibt zurück von wann bis wann das gegebene Gemüse gesät werden kann.

- Mondkalenderdaten
  Um noch einen Web Service mit ins Spiel zu bringen, soll der Agent innerhalb der Saatzeit die besten Termine zum Säen ermitteln. Gartentätigkeiten können auf den Mondkalender abgestimmt werden. So gibt es Tage, die für Tätigkeiten wie säen, ernten, zurückschneiden etc. abhängig vom Stand des Mondes besonders günstig sind. Berücksichtigt wird dabei meist die Mondphase und das Sternzeichen, in dem sich der Mond zu dem Zeitpunkt befindet. Relevant sind vier Sternzeichengruppen mit je drei Sternzeichen und vier korrespondierende Gemüsegruppen: Blattgemüse, Fruchtgemüse, Wurzelgemüse und Blütengemüse.

In Bezug auf das Niveau sollte das System etwa einem Gartenratgeber gleichen und keinen wissenschaftlichen Anspruch besitzen.
Der Rahmen der Studie wurde sehr eng gehalten. Der Agent findet auf den annotierten Seiten nur Informationen zu drei Gemüsesorten, über die er befragt werden kann: Kopfsalat, Tomate und Buschbohne. Die Gemüsesorten sind so gewählt, dass sie möglichst unterschiedliche Wachstumsbedingungen brauchen und zu möglichst unterschiedlichen Kategorien (Blattgemüse, Fruchtgemüse etc.) gehören.
Natürlich ließe sich der Rahmen beliebig ausdehnen. Ein wirklich nützliches System müsste die ganze Gemüsepalette, die hier zu Lande üblich ist, kennen. Auch wären Informationen über Pflege, Ernte, Schädlinge etc. sinnvoll.

3.3 Datenbasis

Das „Semantic Web“, in dem sich der Vegi-Agent bewegt, besteht aus einem Web Service und insgesamt fünf HTML-Seiten aus dem WWW, die im Folgenden kurz beschrieben werden sollen. Ihre Glaubwürdigkeit wird hier vorausgesetzt.

3.3.1 Gartenatelier

Das Gartenatelier[10] ist in einen umfangreichen Hypertext über sämtliche Lifestyle-Themen wie Wohnen, Kochen, Garten etc. eingebettet. Die hier verwendete Gartenseite beinhaltet hauptsächlich Links zu unterschiedlichen Gartenthemen und zu den einzelnen Gemüsesorten. Inhaltliche Informationen enthält sie sonst nicht. Der Agent soll von dieser Seite aus die relevanten Links verfolgen. Relevant ist zum einen die Seite über Kopfsalat [11], die Seite über Buschbohne[12] und die Seite über Tomate[13]. Erst auf diesen Seiten findet der Vegi-Agent inhaltliche Informationen zu der Gemüsesorte. Die Seiten bestehen aus mäßig strukturiertem Volltext mit vielen Überschriften im Verhältnis zu eher knappem Text, sodass man als menschlicher Benutzer rasch einen Überblick über den Inhalt bekommt. Sie bieten Angaben zu Standort, Bodenbeschaffenheit, Saatzeit und einigen weiteren Themen, die der Agent aber nicht berücksichtigt. Abbildung 3.1 zeigt einen Ausschnitt der Kopfsalatseite.

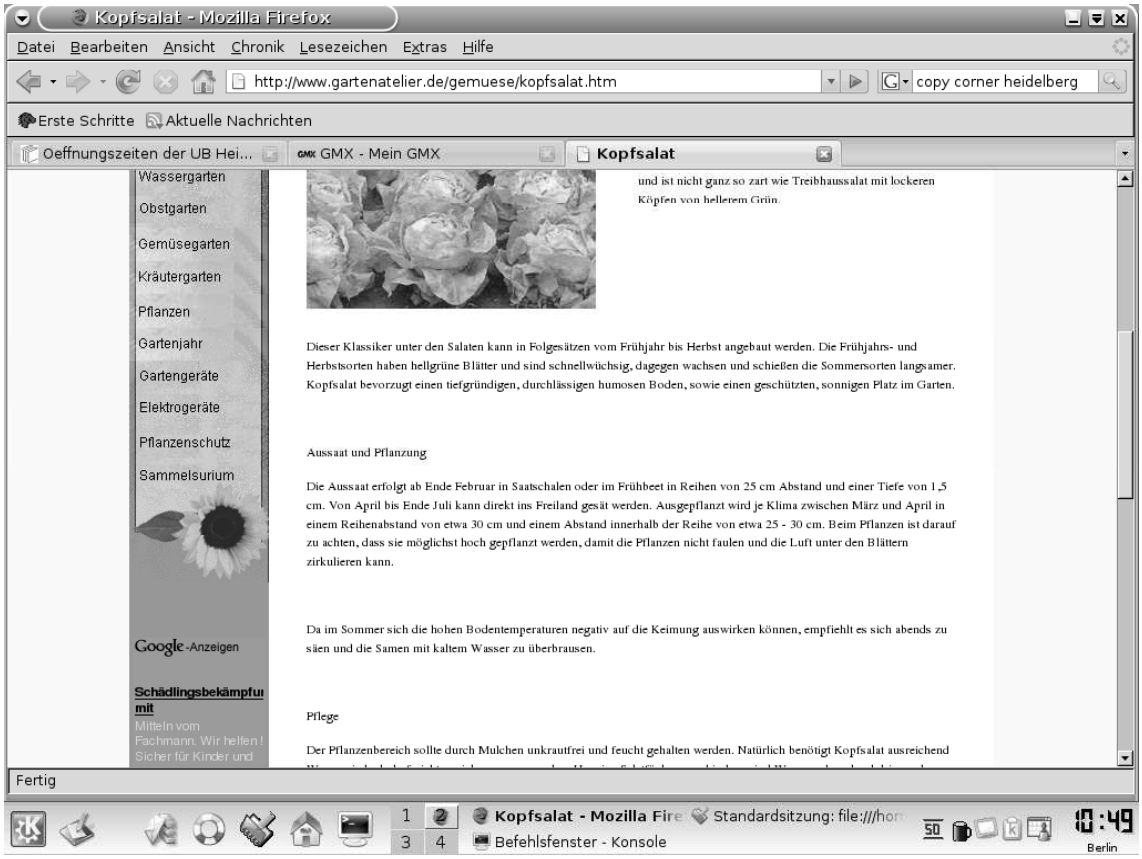
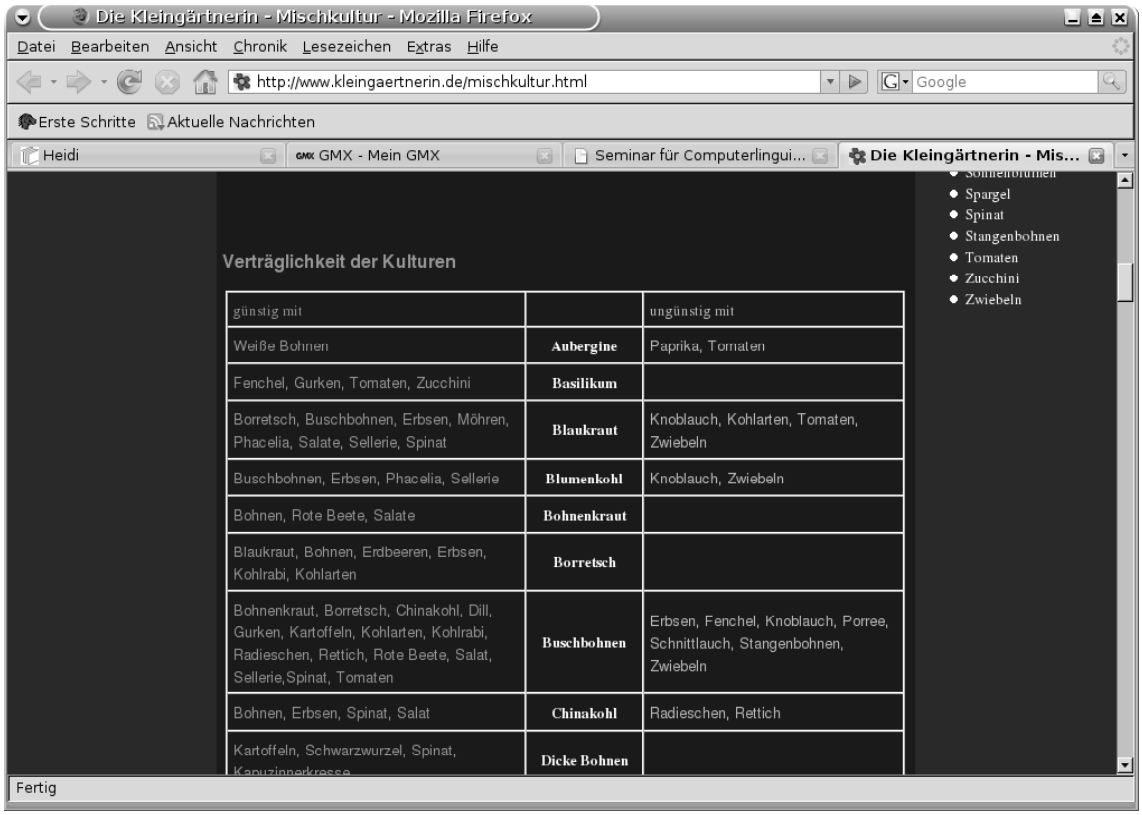


Abbildung 3.1: Die Kopfsalatseite des Gartenateliers

[10] http://www.gartenatelier.de/gemuesegarten.htm
[11] http://www.gartenatelier.de/gemuese/kopfsalat.htm
[12] http://www.gartenatelier.de/gemuese/buschbohnen.htm
[13] http://www.gartenatelier.de/gemuese/tomate.htm

3.3.2 Die Kleingärtnerin

Die gewählte Webseite ist Teil eines komplexen Hypertextes für Heimgärtner und enthält sehr ausführliche Informationen über Mischkulturen[14]. Neben allgemeinen Erklärungen in natürlichsprachigem Text enthält sie eine umfangreiche Tabelle über gute und schlechte Nachbarn einer Pflanze. Diese Tabelle muss der Vegi-Agent bei seiner Informationsbeschaffung auswerten. Die Seite ist ausschnittsweise abgebildet in Abbildung 3.2



günstig mit		ungünstig mit
Weiße Bohnen	**Aubergine**	Paprika, Tomaten
Fenchel, Gurken, Tomaten, Zucchini	**Basilikum**	
Borretsch, Buschbohnen, Erbsen, Möhren, Phacelia, Salate, Sellerie, Spinat	**Blaukraut**	Knoblauch, Kohlarten, Tomaten, Zwiebeln
Buschbohnen, Erbsen, Phacelia, Sellerie	**Blumenkohl**	Knoblauch, Zwiebeln
Bohnen, Rote Beete, Salate	**Bohnenkraut**	
Blaukraut, Bohnen, Erdbeeren, Erbsen, Kohlrabi, Kohlarten	**Borretsch**	
Bohnenkraut, Borretsch, Chinakohl, Dill, Gurken, Kartoffeln, Kohlarten, Kohlrabi, Radieschen, Rettich, Rote Beete, Salat, Sellerie,Spinat, Tomaten	**Buschbohnen**	Erbsen, Fenchel, Knoblauch, Porree, Schnittlauch, Stangenbohnen, Zwiebeln
Bohnen, Erbsen, Spinat, Salat	**Chinakohl**	Radieschen, Rettich
Kartoffeln, Schwarzwurzel, Spinat, Kapuzinerkresse	**Dicke Bohnen**	

Abbildung 3.2: Mischkulturen

3.3.3 Evolutio Rodurago

Der Web Service Evolutio Rodurago[15] bietet einen Algorithmus an, der von Beginn der Zeitrechnung bis ins Jahr 2400 Mondkalender berechnet. Dafür gibt man den gewünschten Monat, das gewünschte Jahr und geographische Daten ein und erhält den Kalender für den Monat. Zu jedem Tag macht der Kalender Angaben über die Mondphase, Mondauf- und -untergang, Sonnenauf- und -untergang und das Sternzeichen, in dem sich der Mond befindet. Einen Ausschnitt des Kalendermonats März

[14] http://www.kleingaertnerin.de/mischkultur.html

[15] http://www.rodurago.de/index.php?month=3&year=2007&geodata=52.33%2C13.22%2C1&site=details&link=calendar

2007 findet sich in Abbildung 3.3

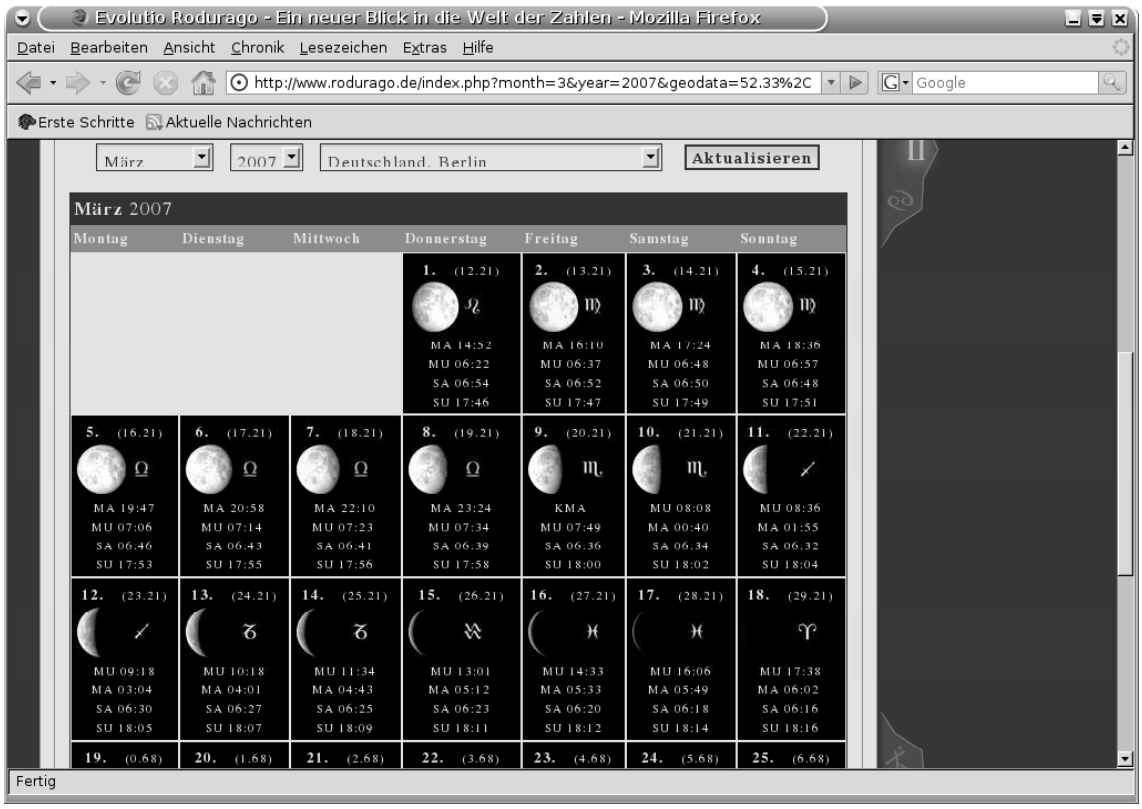


Abbildung 3.3: Mondkalender

Der Nachteil an dem Mondkalender ist, dass er die Mondzustände nur tageweise angibt und nicht nach Uhrzeiten genau. Da der Wechsel von einem Sternzeichen in das nächste zu jeweils unterschiedlichen Uhrzeiten stattfindet, rundet der Kalender großzügig auf oder ab, was zu erheblichen Ungenauigkeiten führt, die in der Studie aber nicht mitberücksichtigt wurden.

3.4 Aufgabenbeschreibung

Der Agent bekommt als Eingabe eine Gemüsesorte in Form eines Strings, den der Benutzer direkt in die Kommandozeile eingeben kann. Die Ausgabe ist eine Tabelle, die die gesuchten Antworten auflistet (siehe Abbildung 3.4). Wie der Agent zu der geforderten Information gelangt, bleibt dem Benutzer verborgen.

3.5 Systemarchitektur

Abbildung 3.5 verdeutlicht den modularen Aufbau des Agenten. Neben Ein- und Ausgabe benützt der Agent folgende Komponenten:

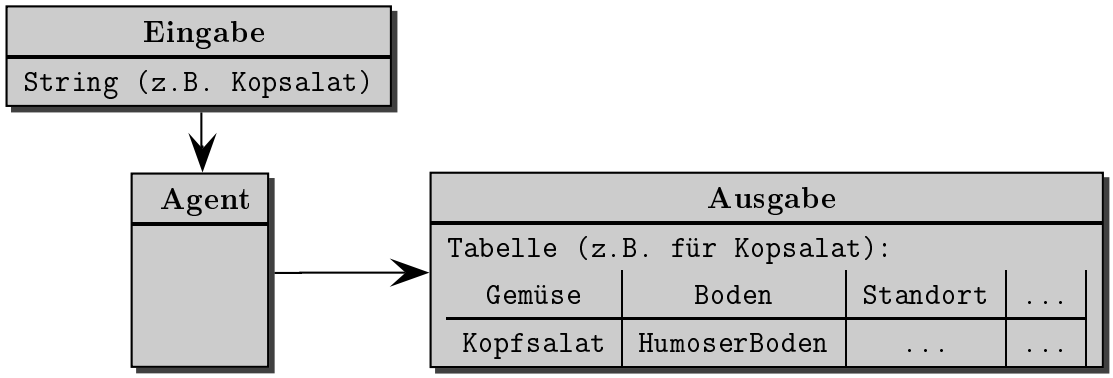


Abbildung 3.4: Ein- und Ausgabestruktur des Agenten

- die Ontologien, von denen der Agent mit der Abfragesprache SPARQL Informationen bezieht.
- die annotierten Webseiten als RDF-Dokumente, die ebenfalls mit SPARQL abgefragt werden.
- den Web Service, der wiederum aus mehreren Teilkomponenten besteht:
  - eine OWL-S-Beschreibung des Mondkalenders, die dem Agenten Informationen liefert, was der Web Service kann und wie auf ihn zugegriffen wird. Sie enthält unter anderem den URI des WSDL-Dokuments.
  - eine WSDL-Beschreibung des Mondkalenders, die der Agent mit Hilfe der WSDL-Tools liest. Mit diesen Informationen wird der Web Service ausgeführt.
  - die HTML-Seite des ausgeführten Mondkalenders wird an den XSLT-Prozessor übergeben.
  - das Stylesheet wird von der OWLS-Beschreibung geliefert.
  - Mit Hilfe des Stylesheets wird das HTML-Dokument in ein RDF-Dokument transformiert, das der Agent mit SPARQL abfragen kann.

Die Fragen, die das System beantworten kann, sind in dem Agenten aufgelistet.

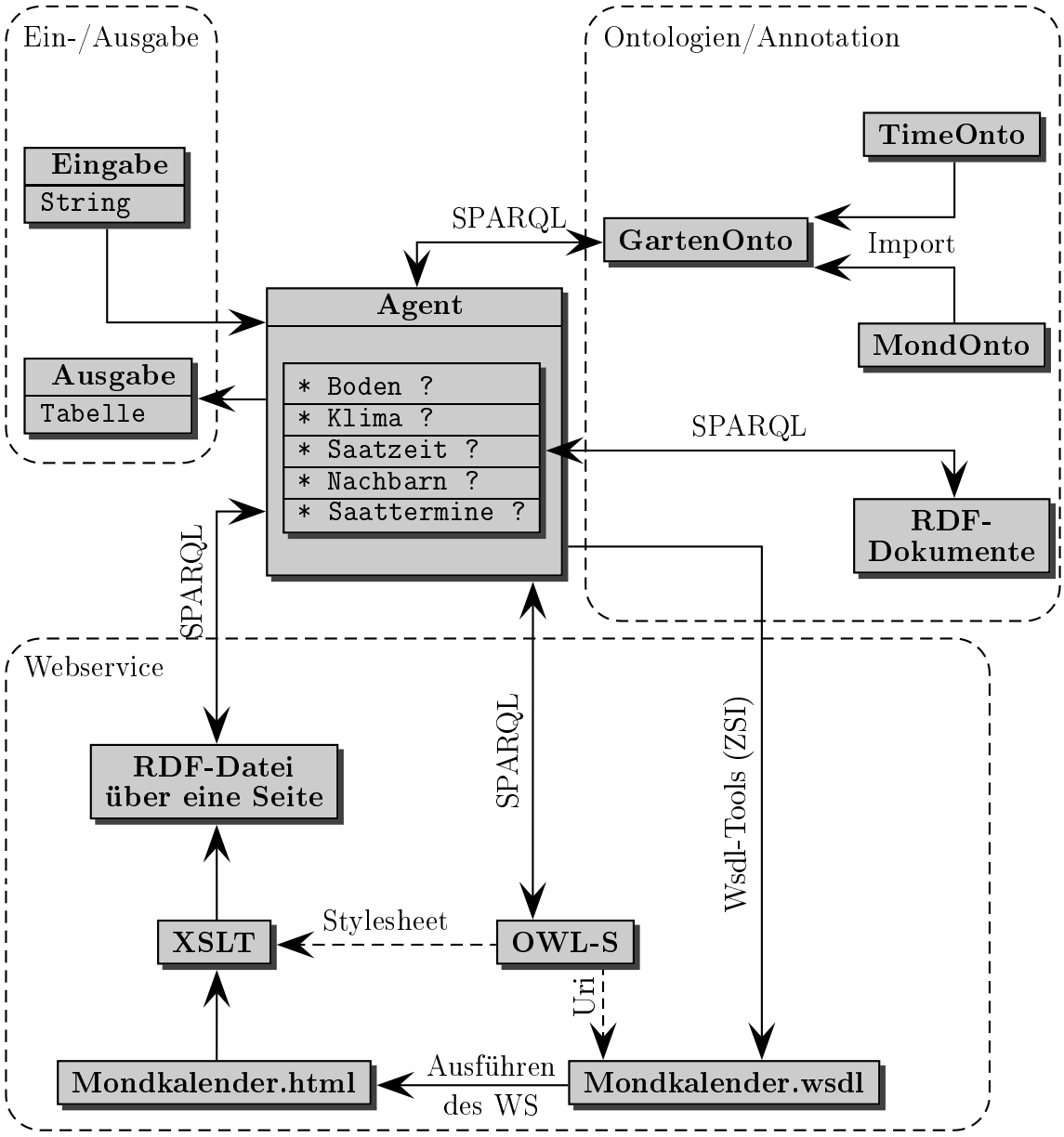


Abbildung 3.5: Systemarchitektur

4 Die Ontologien

Um die geforderte Information liefern zu können, braucht der Agent Informationen aus unterschiedlichen Bereichen. Zum einen muss er sich natürlich über die Pflanzen selbst auskennen. Ferner braucht er Informationen über Räumlichkeiten, um zu wissen, dass die Pflanzen an einem Ort wachsen. Da es sich bei allen Gärtnertätigkeiten um zyklische Prozesse handelt, ist Wissen über Zeitverhältnisse nötig. Schließlich braucht der Agent noch einige astrologische Informationen, damit er über den Mondkalender Aussagen machen kann.
Da es sich hier um recht unterschiedliche Wissensgebiete handelt, wurden mehrere Ontologien erstellt bzw. übernommen. Eine Gartenontologie über die Pflanzen und die Ortsverhältnisse musste von Grund auf neu entworfen werden. Ebenso die Mondkalender-Ontologie, die aber minimal gehalten wurde und unmittelbar auf den verwendeten Mondkalender zugeschnitten ist.
Das W3C hat eine sehr detailierte Ontologie über Zeitverhältnisse entwickelt, die mit einigen Erweiterungen übernommen wurde. Außerdem wurde eine ebenfalls vom W3C entwickelte Ontologie für Web Services miteinbezogen, die den Service semantisch beschreibt.
Auf die einzelnen Ontologien soll im Folgenden näher eingegangen werden.

4.1 Die Gartenontologie

Die Ontologie beschreibt das Wissensgebiet der Hobby-Gärtnerei. Sie ist für Freizeit-Gärtner konzipiert, die ihren eigenen Garten bestellen möchten und dafür Know-How und Tipps brauchen. Sie ist nicht für den professionellen Gartenbau gedacht, für den sehr viel mehr Detailwissen nötig wäre. Für die Studie wurde der Onotlogie bewusst sehr enge Grenzen gesetzt. Sie beschränkt sich lediglich auf Informationen über die Aussaat der Pflanzen.

4.1.1 Planung der Ontologie

Bevor man eine Ontologie neu erstellt, lohnt es sich, nach Ontologien zu suchen, die das Wissensgebiet oder wenigstens einen Teil davon abdecken. Zwei freie Ontologie-Bibliotheken im Web sind z.B. die Ontolingua Ontology Library[16] oder die DAML

[16]Vgl. Ontolingua: http://www.ksl.stanford.edu/software/ontolingua/

Ontology Library[17]. Eine Gartenontologie hat sich aber auch nach intensiver Suche nicht finden lassen. Ganz generell scheint der Freizeitbereich noch völlig unerschlossen zu sein. Auch eine Anlehnung an die botanische Nomenklatur hat sich nicht angeboten. Einerseits ist ihre Granularität für einen Freizeitgärtner viel zu hoch. Denn ob ein Kopfsalat zu der Klasse der Zweikeimblättrigen oder zu der Unterabteilung der Bedecktsamer gehört, ist nicht von Relevanz. Andererseits kommen viele Konzepte, die für den Garten wichtig sind, wie z.B. Gemüse, Blume etc. überhaupt nicht vor. Deshalb wurde eine Gruppierung in Anlehnung an einen Garten-Ratgeber[18] gewählt. Denkbar wäre eine Ontologie, die beide Hierarchien beinhaltet, indem den Pflanzen mehrere Oberklassen zugeordnet werden. Die Ontologie wäre damit für sehr viel mehr Anwendungen geeignet, wäre aber auch sehr viel schwieriger zu durchschauen.
Ein Charakteristikum der Domäne ist die vage und kaum definierte Begrifflichkeit. Beispielsweise gibt es große Überlappungen in den Kategorien. So gilt Borretsch meist als Küchenkraut, könnte wegen seiner hübschen Blüten aber auch in die Kategorie der Zierpflanzen fallen. Auch sind ungenaue Zeitangaben wie z.B. *Ende Februar* sehr gängig und durchaus effizient, weil der Gärtner weiß, dass er Witterungsbedingungen, seine geographische Lage etc. miteinbeziehen muss. Diese Vagheiten müssen in der Ontologie berücksichtigt werden.
Es stellt sich ferner die Frage, was die Ontologie überhaupt wissen muss. Muss sie z.B. wissen, dass der April vor dem Mai liegt und dass es den 40. Februar nicht gibt? Da die verarbeitenden Programme in Python oder anderen Programmiersprachen diese Information haben, ist die Kodierung in der Ontologie überflüssig und daher in der Time-Ontologie auch nicht vorhanden.
Vor der Entwicklung der Ontologie wurden einige Kompetenzfragen formuliert, die einerseits dazu dienen, den Umfang der Ontologie zu umreißen, und die andererseits zum Testen verwendet wurden. Sie zeigen an, ob die Ontologie genügend Information enthält. Folgende Kompetenzfragen wurden gestellt:

- Welche Pflanzen vertragen sich mit Tomaten?
- Wann sät man am besten Kopfsalat?
- Welches Gemüse verträgt mageren Boden?
- Welchen Standort braucht Kopfsalat?
- In welcher Mondphase sät man Buschbohnen?
- In welchen Sternzeichen sät man Kopfsalat?

[17] Vgl. DAML Ontology Library: http://www.daml.org/ontologies/
[18] Vgl. Kreuter 2001

Bei der Namensgebung innerhalb der Ontologie sind folgende Konventionen eingehalten worden: Klassen beginnen mit einem Großbuchstaben, dann fogen Kleinbuchstaben; jedes weitere Wort wird mit einem Großbuchstaben abgesetzt, z.B. `PhaseOfMoon`. Eigenschaften beginnen mit einem Kleinbuchstaben und möglichst mit den Verben *is* oder *has*, z.B. `isSeeded, hasClimate`. Instanzen, die im engeren Sinne aber nicht mehr zur Ontologie gehören, folgen dem Namensmuster der Klassen. Die Namen innerhalb der Ontologie sind einheitlich englisch, da englischsprachige Ontologien importiert und erweitert wurden. Die Instanzen, die aus der Annotation deutscher Webseiten resultieren, haben dagegen deutsche Namen.
Es wurde darauf geachtet, den Klassen und Eigenschaften möglichst selbstsprechende Namen zu geben, was den Vorteil hat, dass bereits aus der URI erkenntlich ist, welche Instanzen in einer Klasse zu erwarten sind bzw. wie eine Eigenschaft angewendet werden muss. Besonders ein Annotator, der die Ontologie weniger gut kennt, ist damit weniger auf die Dokumentation angewiesen.

4.1.2 Die Entwicklung der Ontologie

Im ersten Entwicklungsschritt wurde ein Brain-Storming über die wichtigsten Begriffe der Domäne gemacht, wie z.B. *Pflanze, Gemüse, Blattgemüse, Fruchtgemüse, sonnig, schattig, säen* etc. Diese zunächst wahllos zusammengestellten Begriffe wurden anschließend in folgenden Arbeitsschritten bearbeitet:

- Klassen der Ontologie bestimmen
- Die Klassen in eine hierarchische Struktur überführen
- Eigenschaften bestimmen und erlaubte Werte für die Eigenschaften definieren
- Die Werte der Eigenschaften mit Instanzen füllen

Idealerweise sollten letztendlich alle Begriffe untergebracht sein. Noy/McGuinness (2001, S.4) betonen aber, dass diese Schritte wiederholt werden müssen, so lange eine Ontologie in Gebrauch ist, weil Überarbeitungen und Korrekturen immer wieder nötig werden. Sie bezeichnen deshalb Ontologieentwicklung als einen notwendigerweise iterativen Prozess.
Der oberste Knoten in der Hierarchie einer OWL-Ontologie ist die Klasse `Thing`, was heißt, dass jede Instanz der Ontologie Mitglied dieser Klasse ist. Im Gegensatz dazu steht die Klasse `Nothing`, die leere Klasse. Diese Klasse hat nur Unterklassen, wenn diese inkonsistent sind, wenn diese Klasse also in keinem Fall Instanzen haben kann. Auf der ersten Ebene unter der Klasse `Thing` wurden der Übersichtlichkeit halber die vier sehr allgemeine Klassen `PlantThing, LocationThing, TimeThing` und `MoonThing` angelegt, die die Hauptbereiche Pflanze, Zeit, Ort und Mond trennen.

Anderenfalls würden auf der oberen Ebene knapp zwanzig Klassen liegen, die in den Semantic Web-Werkzeugen alphabetisch und nicht thematisch geordnet dargestellt werden, was die Bearbeitung erschweren würde.

4.1.3 PlantThing

Alles, was mit den Pflanzen selbst zu tun hat, ist in dem Hauptzweig `PlantThing` untergebracht. Implementiert wurden die Teile einer Pflanze und die Taxonomie der Pflanze wie in Abbildung 4.1 dargestellt. Die Ellipsen mit Punkten deuten an, dass nicht alle Klassen abgebildet sind.
Mit den Eigenschaften `inGoodNeighbourhoodWith` bzw. `inBadNeighbourhoodWith` können die Instanzen einer `CultivatedPlant` in Bezug auf die Nachbarschaftsverhältnisse miteinander in Beziehung gesetzt werden. Der Definitionsbereich wurde mit der Klasse `CultivatedPlant` belegt, weil davon ausgegangen wird, dass die Eigenschaften von Wildpflanzen für Gärtner nicht relevant sind. Die Begründung gilt auch für die im Folgenden beschriebenen Eigenschaften.

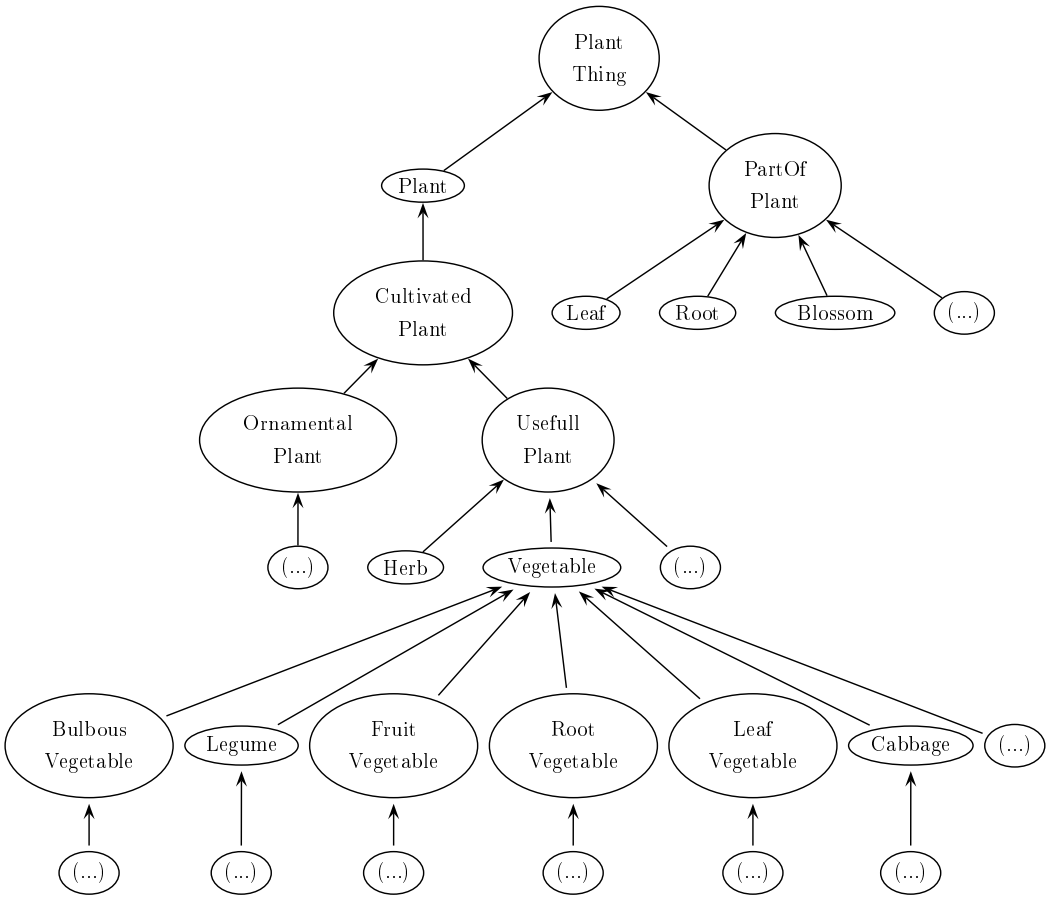


Abbildung 4.1: Baumstruktur der Klasse PlantThing und seiner Kindknoten

Die Eigenschaften `isSeededIndoor` und `isSeededOutdoor` beschreiben, wann eine Pflanze wohin gesät werden muss. Sie haben jeweils zwei Wertebereiche, die zum einen den passenden Ort (`IndoorLocation` bzw. `OutdoorLocation`) und zum anderen die passende Säzeit (`SeedingInterval`) angeben.
Die Gartenontologie ist mit der Mondontologie über die Eigenschaft `bestSeedingTime` verbunden. Diese Eigenschaft hat als Definitionsbereich `CultivatedPlant` und die Wertebereiche `PhaseOfMoon` und `MoonInSignOfZodiac`. Mit dieser Verbindung kann ausgedrückt werden, bei welchem Mondstand in Bezug auf die Mondphase und das Tierkreiszeichen eine Pflanze am besten gesät wird. Die beste Säzeit ist also die Schnittmenge der entsprechenden Mondphase mit den drei entsprechenden Tierkreiszeichen-Phasen; oder etwas formaler:

$$Mondphase \cap (Sternzeichen1 \cup Sternzeichen2 \cup Sternzeichen3)$$

Die Eigenschaft `bestSeedingTime` bezieht sich auf die astrologischen Daten und ist für die Klassen `FruitVegetable`, `LeafVegetable`, `BlossomVegetable` und `RootVegetable` definiert. Im Quellcode sieht die Klasse `FruitVegetable` aus wie in Listing 4.1. Es werden mehrere anonyme Klassen gebildet: eine Klasse besteht aus der Vereinigungsmenge der drei Sternzeichenphasen und eine zweite Klasse bildet die Schnittmenge der Mondphase mit den Sternzeichenphasen.

```
1  <owl:Class rdf:about="&garden;FruitVegetable">
2    <rdfs:subClassOf>
3      <owl:Class rdf:about="&garden;Vegetable"/>
4    </rdfs:subClassOf>
5    <rdfs:subClassOf>
6      <owl:Restriction>
7        <owl:onProperty rdf:resource="&moon;bestSeedingTime"/>
8        <owl:someValuesFrom>
9          <owl:Class>
10           <owl:intersectionOf rdf:parseType="Collection">
11             <owl:Class rdf:about="&moon;WaxingMoon"/>
12             <owl:Class>
13               <owl:unionOf rdf:parseType="Collection">
14                 <owl:Class rdf:about="&moon;MoonInAries"/>
15                 <owl:Class rdf:about="&moon;MoonInLeo"/>
16                 <owl:Class rdf:about=
17                     "&moon;MoonInSagittarius"/>
18               </owl:unionOf>
19             </owl:Class>
20           </owl:intersectionOf>
21         </owl:Class>
```

```
22        </owl:someValuesFrom>
23      </owl:Restriction>
24    </rdfs:subClassOf>
25 </owl:Class>
```

Listing 4.1: Definition der Klasse `FruitVegetable`

4.1.4 LocationThing

Der hier beschriebene Ontologie-Zweig bezieht sich auf die Räumlichkeit, in denen die Pflanzen wachsen. Hauptsächlich sind hier zwei Unterscheidungen nötig: Standorte drinnen und Standorte draußen, die dann noch weiter in Unterklassen wie `Garden, VegetableBed` etc. aufgegliedert werden können. Ferner musste modelliert werden, dass Örtlichkeiten unterschiedliche Qualitäten besitzen in Bezug auf das Klima, z.B. *sonnig, schattig, windig* etc. und in Bezug auf Eigenschaften des Bodens wie z.B. *fruchtbar, mager, locker* etc., siehe Abbildung 4.2. Die Eigenschaft `hasClimate` bzw. `hasGround` verbindet einen Pflanzenort mit den Klassen `ClimateAtLocatin` bzw. `GroundAtLocation` und deren Unterklassen wie `SunnyClimate` etc. bzw. `FertileGround` etc., die die Qualitätsunterschiede der Orte kennzeichnen.
Die Verbindung zu den Pflanzen wird durch die Eigenschaften `needsGround` und `needsClimate` hergestellt. Mit ihnen kann man einer Kulturpflanze die richtigen Standortbedingungen zuweisen. Der Definitionsbereich ist also `CultivatedPlant` und die Wertebereiche sind `GroundAtLocation` bzw. `ClimateAtLocation`.

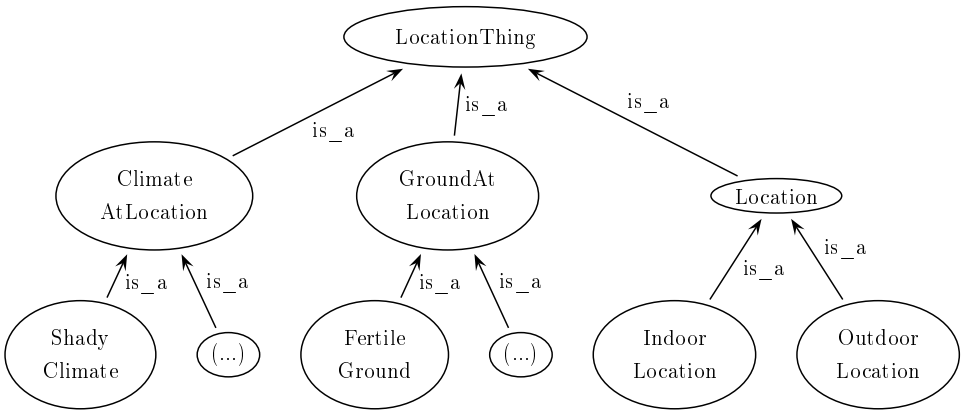


Abbildung 4.2: Baumstruktur der Klasse `LocationThing` und ihrer Kindknoten

4.1.5 Design-Entscheidungen

Bei der Modellierung einer Ontologie gibt es keine festen Regeln, die ein Richtig oder Falsch anzeigen würden. Es gibt immer verschiedene brauchbare Alternativen,

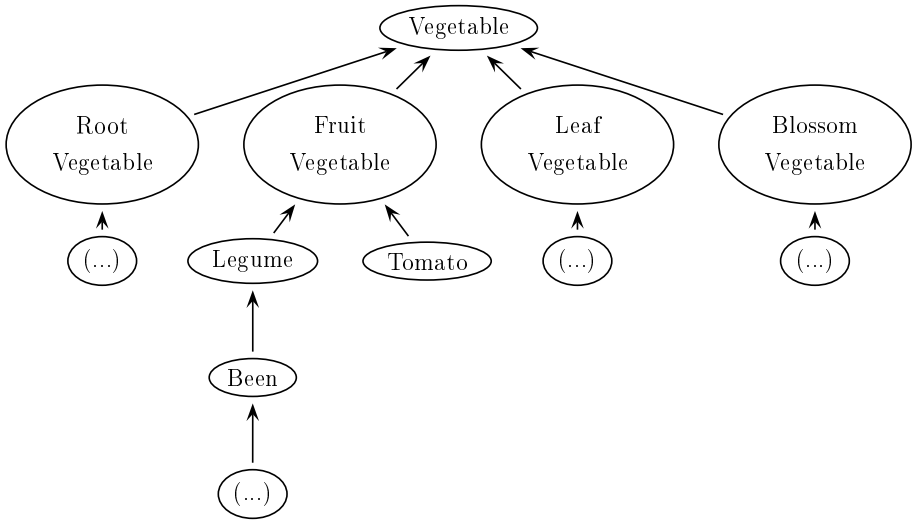


Abbildung 4.3: Alternative1 zu den Kindknoten der Klasse `Vegetable`

die gegeneinander abgewogen werden müssen. Die beste Lösung wird meistens durch die konkrete Anwedung nahegelegt. Ferner ist darauf zu achten, dass die Ontologie eine Abbildung der Realität darstellt und die Konzepte der Ontologie diese Realität widerspiegeln sollen.[19].

Einige konkrete Design-Entscheidungen werden im Folgenden kurz skizziert: Eine Frage ist z.B. die Einordnung nach Unterklasse, Oberklasse, Geschwister. Ist die Klasse Hülsenfrüchte ein Kindknoten von Gemüse oder von Fruchtgemüse? Entscheidet man sich für Unterklasse unter Fruchtgemüse, müsste man noch einen Oberbegriff für andere Fruchtgemüsesorten wie Tomaten und Auberginen einführen, weil

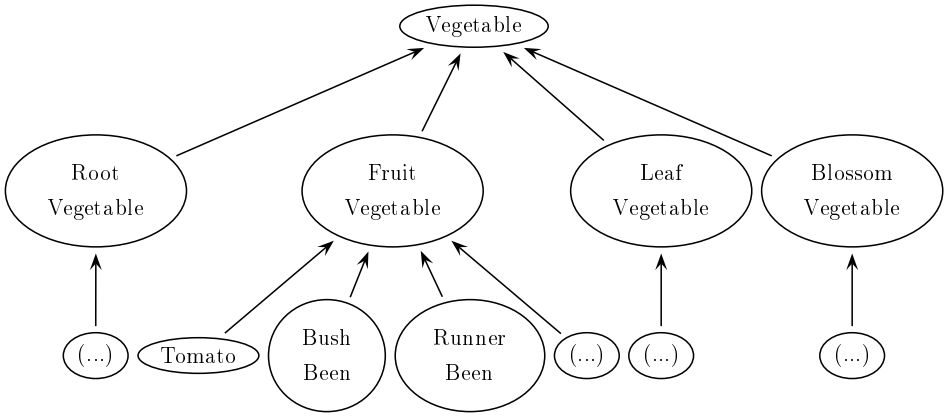


Abbildung 4.4: Alternative2 zu den Kindknoten der Klasse `Vegetable`

[19]Noy, 2004, S. 4

es sonst zu unterschiedlich spezifischen Geschwiserknoten kommt (siehe Abbildung 4.3). Bei dieser Anordnung kämen nur die Klassen *Frucht-, Wurzel-, Blüten-,* und *Blattgemüse* als unmittelbare Kindknoten von der Klasse *Gemüse* vor, was den Gemüsekategorien des Mondkalenders unmittelbar entsprechen würde.
Eine weitere Möglichkeit wäre die Unterklasse Hülsenfrüchte wegzulassen und die Sorten wie *Buschbohne, Stangenbohne* etc. direkt unter Fruchtgemüse zu legen (siehe Abbildung 4.4). Die Folge wäre eine sehr grobe Granulation, wodurch Eigenschaften aller Bohnen wie z.B. Nachbarschaftsverhältnisse zu anderen Pflanzen jeder einzelnen Bohnenart zugewiesen werden müssten, statt eine Eigenschaft mit dem Wertebereich *Bohne* zu definieren.
In der hier verfolgten Alternative liegen die Klassen *Hülsenfrüchte* und *Fruchtgemüse* als Geschwisterknoten in Anlehnung an den Gartenratgeber[20] auf einer Ebene (siehe Abbildung 4.5). Um dennoch der Klasse *Hülsenfrüchte* die Mondkalender relevanten Eigenschaften der Klasse *Fruchtgemüse* zu vererben, wird diese als zweite Oberklasse zugewiesen. Die Klasse Hülsenfrüchte erbt damit die Eigenschaften von *Gemüse* und *Fruchtgemüse.*

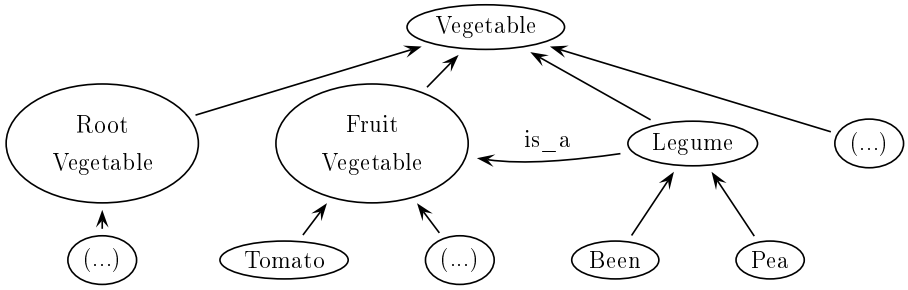


Abbildung 4.5: Alternative3 zu den Kindknoten der Klasse `Vegetable`

Eine weitere Frage stellt sich bei der Granulationstiefe der Ontologie: Wo hören die Klassen auf, wo beginnen die Instanzen? Gibt es noch eine Klasse `Tomato` oder ist dies bereits eine Instanz? Bei dem Konzept `Tomato` könnte man noch stärker in die Tiefe gehen und z.B. eine Klasse mit unterschiedlichen Tomatensorten bilden, was für das Wissensgebiet durchaus von Relevanz sein kann. Da aber die Pflanzzeiten der einzelnen Züchtungen auf den bearbeiteten Webseiten nicht angegeben sind, ist eine solch feine Granulation für diese Anwendung nicht notwendig. Als grobe Faustregel gilt: das spezifischste Konzept der Wissensbasis bildet die Instanz. Einen Anhaltspunkt bieten auch die eingangs gestellten Kompetenzfragen. Das spezifischste Konzept, das Antworten auf die Fragen liefert, ist ein guter Kandidat, Instanz in der Wissensbasis zu sein[21]. Die Gartenontologie verfolgt diese Faustregel bisher noch

[20] Kreuter, 2001
[21] Vgl. Noy/McGuinnes, 2001, S. 18

nicht, weil zunächst davon ausgegangen wurde, dass die Instanz das konkrete Gemüse ist, das im Garten wächst oder auf der Webseite steht. Das hat dazu geführt, dass sich Instanzen und Klassen bei der Benennung nur noch durch die Sprache unterscheiden. Eine terminierende Klasse der englischsprachigen Gartenontologie lautet z.B. `Tomato`, während die Instanz, die aus der deutschsprachigen Annotation hervorgeht, `Tomate` heißt, was sicherlich keine sehr glückliche Lösung ist.
Einer kritischen Überprüfung hält die Gartenontologie momentan noch nicht stand. Die Subklassen sind zwar streng taxonomisch, die Geschwisterknoten sind aber teils noch recht heterogen und haben nicht unbedingt das gleiche Niveau. Auch ist die Ontologie noch viel zu wenig getestet. Viele Klassen kommen innerhalb der Studie nicht zur Anwendung und könnten im Prinzip eliminiert werden. Sie sind aber im Hinblick auf Erweiterungen implementiert, da Veränderungen der Ontologie Auswirkungen auf die Annotation der Webseiten und die Implementierung des Agenten haben, wie später noch deutlich wird.

4.2 Die Time-Ontologie

Die Time-Ontologie in OWL ist ein Arbeitsentwurf des W3C vom September 2006[22]. Entwicklungsgeschichtlich ist sie eine Vereinfachung der DAML-Time-Ontologie[23] und wurde mit dem Ziel entwickelt, einen möglichst einfachen Zugang zu den grundlegenden temporalen Konzepten und Relationen zu bieten. Die Time-Ontologie beschreibt ein Vokabular, mit dem topologische Fakten wie Zeitpunkte und Zeitintervalle sowie Angaben zu Dauer, Datum und Uhrzeit ausgedrückt werden können[24]. Sie ist kompatibel mit OWL-S, damit sie auch für Web Services gut zu verwenden ist (siehe Kapitel 4.4).
Im Folgenden wird die Time-Ontologie mit den Klassen und Eigenschaften beschrieben, die Relevanz für die Gartenontologie besitzen.

4.2.1 Die Klasse `TemporalEntity`

Das Grundkonzept der Ontologie ist die Klasse `TemporalEntity` mit den Unterklassen `Interval` und `Instant`. Genau genommen sind die beiden Unterklassen als Equivalenzklasse zu ihrer Oberklasse spezifiziert, was heißt, dass keine anderen Unterklassen zu `TemporalEntity` existieren als diese zwei. `TemporalEntity` hat die Eigenschaften `hasBeginning` und `hasEnd`. Diese Eigenschaften haben als Definitionsbereich `TemporalEntity` und einen `Instant` als Wertebereich, stellen also einen

[22] Vgl. Time-Ontologie: http://www.w3.org/2001/sw/BestPractices/OEP/Time-Ontology
[23] Vgl. DAML-Time: http://www.cs.rochester.edu/ ferguson/daml/
[24] Vgl. Pan/Hobbs 2004, S. 29

bestimmten Zeitpunkt – nämlich Anfangs- oder Endpunkt – einer Zeiteinheit dar. Da prinzipiell Anfangspunkt und Endpunkt zusammenfallen können, hat die Klasse `Interval` eine Unterklasse `ProperInterval`, die so definiert ist, dass Anfangs- und Endpunkt unterschiedliche Zeitpunkte sind, wodurch das Intervall eine Dauer haben muss. Damit ist die Klasse `ProperInterval` disjunkt zur Klasse `Instant`, die man sich als ein Intervall vorstellen könnte, bei der Anfang und Ende in einem Punkt zusammenfallen. Ferner hat die Klasse `TemporalEntity` die Eigenschaft `hasDurationDescription`, mit der die Dauer eines Intervalls definiert werden kann. Es ist eine Designentscheidung der Entwickler, diese Eigenschaft nicht erst in der Klasse `Interval` anzulegen, sondern sie auch an die Klasse `Instant` zu vererben, bei der die Dauer aber immer Null sein muss.
Die Klasse ProperInterval bietet eine Anzahl weiterer Eigenschaften, die hier aber nicht zur Anwendung kamen.

4.2.2 Die Klasse `DateTimeDescription`

Die Klassen `DateTimeDescription` dient dazu, einem Zeitpunkt ein bestimmtes Kalenderdatum zuzuweisen. Dafür gibt es die Eigenschaft `inDateTime` mit einem `Instant` als Definitionsbereich und einer `DateTimeDescription` als Wertebereich. Die Klasse enthält sämtliche Monate als Subklassen, die mit der Eigenschaft `month` ihrem entsprechenden `xsd:gMonth` zugewiesen werden können. Für `February` wäre das z.B. `xsd:gMonth --02`. Ähnlich funktionieren die Datentyp-Eigenschaften `day`, `year` etc., deren Definitionsbereiche ebenfalls eine `DateTimeDescription` ist und deren Wertebereiche xsd-Litterale sein müssen. Die Beschreibung eines Individuums *Sommeranfang* würde beispielsweise aussehen wie in Abbildung 4.6.

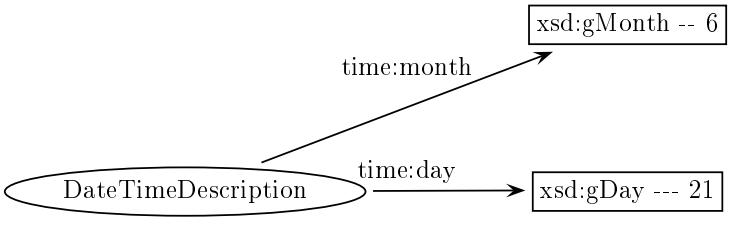


Abbildung 4.6: `DateTimeDescription`: *Sommeranfang*

4.2.3 Die Klasse `DurationDescription`

Mit der Klasse `DurationDescription` können Angaben zur Dauer gemacht werden. Die Dauer eines Intervalls kann sehr unterschiedlich ausgedrückt werden: in Jahren,

Monaten, Wochen, Tagen, Stunden, Minuten und Sekunden. Eine Duration Description hat daher sieben Datentyp-Eigenschaften (`years, months, weeks, ...`), wobei die Werte mit einer `xsd:decimal`-Angabe gefüllt werden. Dadurch kann auch eine Dauer von 2,5 Jahren angeben werden. Eine Dauer von einem Tag, zwei Stunden und 30 Minuten könnte aussehen wie in Abbildung 4.7. Da ein Intervall nur eine

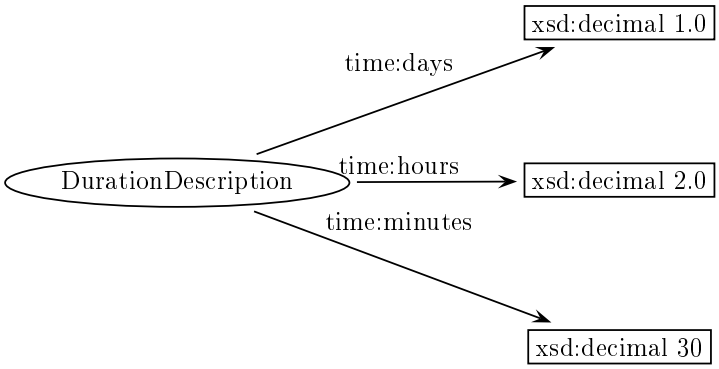


Abbildung 4.7: `DurationDescription`: 1 Tag, 2 Stunden, 30 Minuten

einzige Dauer haben kann, haben alle Datentyp-Eigenschaften eine Kardinalität von maximal eins, was besagt, dass jede dieser Eigenschaften ein Individuum höchstens ein Mal mit einem xsd:decimal-Wert verbinden kann.
Die Klasse `DateTimeDescription` ist, wie oben bereits erwähnt, mit der Klasse `TemporalEntity` über die Eigenschaft `hasDurationDescription` verbunden.

4.2.4 Anpassungen für die Anwendung

Die Klasse `ProperInterval` und `Instant` mussten noch mit vielen Unterklassen bestückt werden, um Saatzeiten für Gemüse darstellen zu können. Dafür wurde zunächst die Klasse `SeedingInterval` als Subklasse von `ProperInterval` gebildet. Etwas umständlich gestalteten sich insbesondere natürlich sprachliche Ausdrücke wie *Ende März, Mitte Juni, nach den Eisheiligen* etc. Um die Bedeutung solch unscharf umrissener Zeitbegriffe für einen Computer interpretierbar zu machen, müssen sie trotz allem an Kalenderdaten festgemacht werden. Deshalb wurde festgelegt, dass der Anfang eines Monats vom 1. bis zum 12. des jeweiligen Monats dauert, Mitte des Monats vom 10. bis zum 20. und Ende des Monat vom 18. bis zum letzten Monatstag. Es musste berücksichtigt werden, dass der letzte Tag im Monat der 28., 30. oder 31. eines Monats sein kann. Diese Werte sind im eigenen Ermessen entstanden und haben noch kein Fundament.
Die Vagheit eines Ausdrucks wie *von Ende März bis Mitte Juni* kommt dadurch zustande, dass es sich bei dem Anfangszeitpunkt und dem Endzeitpunkt genau genommen um keine Zeitpunkte handelt, sondern selbst um Intervalle. Um nun

einen Zeitraum wie den oben genannten zu beschreiben, muss der Anfangszeitpunkt irgendwo innerhalb eines zweiten Intervalls liegen – nämlich in dem Intervall *Ende März*. Genauso verhält es sich mit dem terminierenden Zeitpunkt, der in dem Intervall *Mitte Juni* liegen muss. Um eine solche Struktur umsetzen zu können, sind folgende `Instant`-Subklassen definiert: `InstantFirstPartOfMonth`, `InstantSecondPartOfMonth`, `InstantThirdPartOfMonth31`, `InstantThirdPartOfMonth30` und `InstantThirdPartOfMonthFeb`. Es werden drei Klassen für das Ende eines Monats benötigt, weil dieser Teil bei den unterschiedlichen Monaten unterschiedliche Längen hat. Die Klassen bezeichnen einen umgangssprachlichen Zeitpunkt wie *Ende März* und müssen deshalb eine Relation zu dem Intervall haben, das den entsprechenden Teil des Monats darstellt. Diese Verbindung stellt die Eigen-

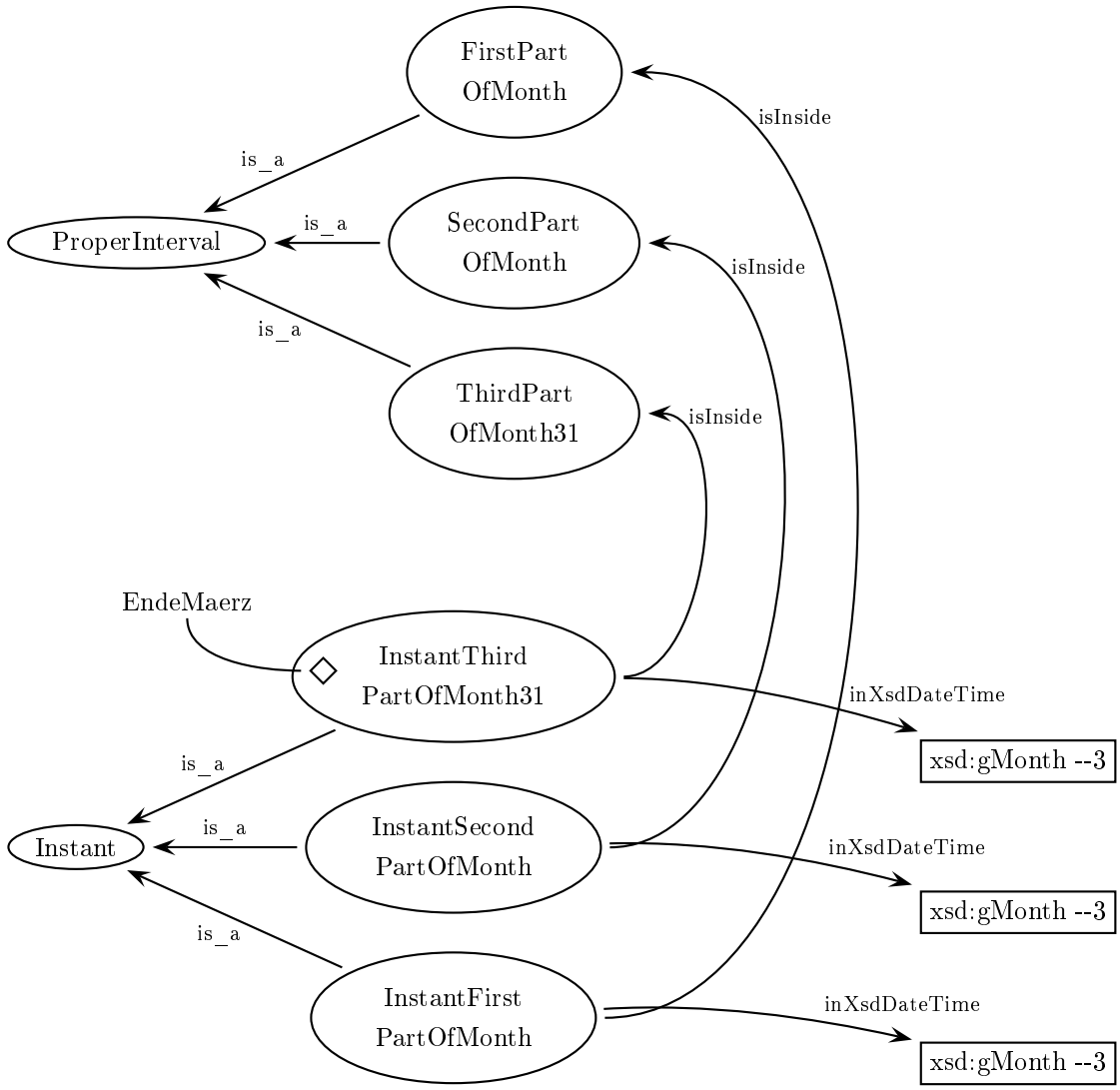


Abbildung 4.8: Beschreibung eines Zeitpunktes wie *Ende März*

schaft `isInside` her, die invers ist zu der Time-Eigenschaft `inside`[25]. Die Intervall-Klassen, die die Monatsteile ausdrücken, wurden allgemein gehalten, um nicht für jeden Monat einen Anfangs-, Mittel- und Endteil als Klasse erstellen zu müssen. Die Information, um welchen speziellen Monat es sich handelt, muss deshalb durch eine `inDateTime`-Relation zu einer `DateTimeDescription` ausgedrückt werden, die den Monat festhält. Abbildung 4.8 zeigt, wie ein natürlich sprachlicher Zeitpunkt wie *Ende März* in die Beschreibung des Monats (März) und die Beschreibung des Monatsteils (Ende) geteilt wird. Das Individuum `EndeMaerz` ist als Raute dargestellt. Desweiteren gibt es die `ProperInterval`-Subklassen `FirstPartOfMonth, SecondPartOfMonth, ThirdPartOfMonth31, ThirdPartOfMonth30` und `ThirdPartOfMonthFeb`. Diese Klassen, die die Monatsteile ausdrücken, haben Relationen zu bestimmten Zeitpunkten, die den Anfang und das Ende jedes Teils definieren. Die Zeitpunkte sind in den `Instant`-Subklassen als `BeginnOfFirstPartOfMonth, EndOf-`

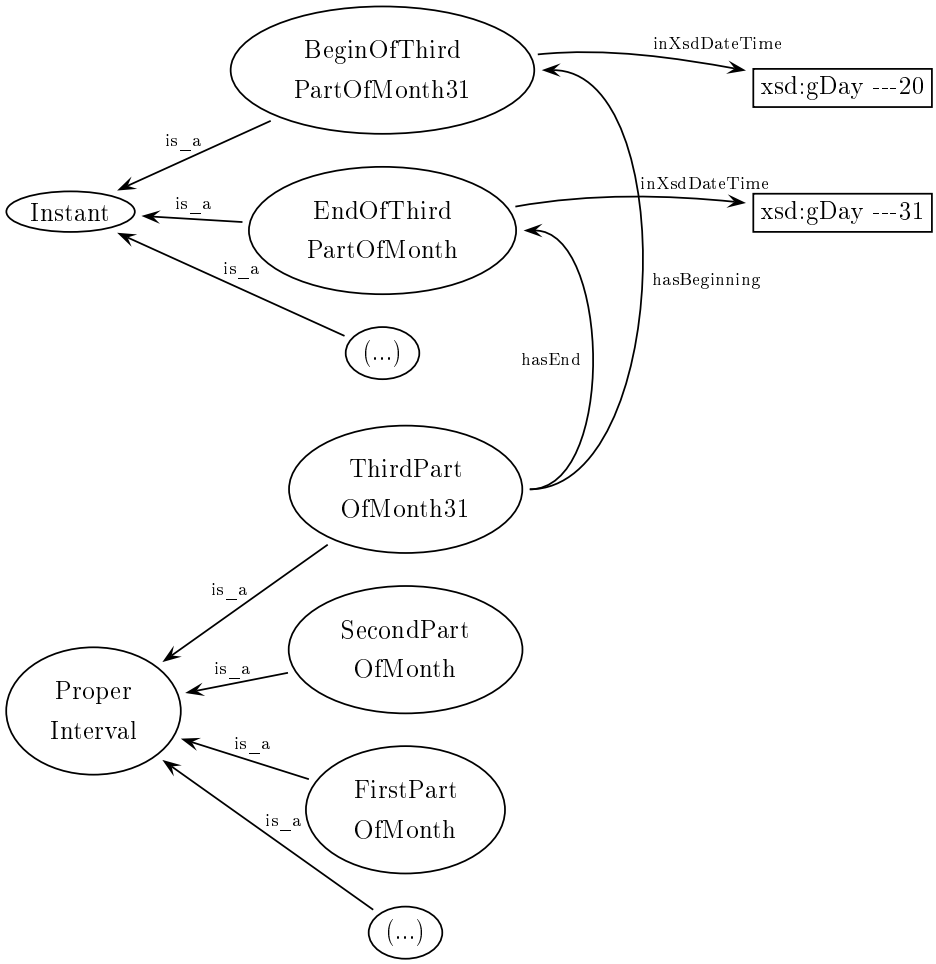


Abbildung 4.9: Beschreibung der Monatsteile

[25] Der Name `inside` ist meines Erachtens etwas missverständlich, da die Eigenschaft ein Intervall als Definitionsbereich und einen Zeitpunkt als Wertebereich hat. Sie ist demnach zu lesen als: Interval X `hasInteriorPointInside` Instant Y.

`FirstPartOfMonth, BeginOfSecondPartOfMonth` etc. realisiert. Diese Zeitpunkte werden dann in einer `DateTimeDescription` einem bestimmten Monatstag zugeordnet. Für `EndeMaerz` wäre das `xsd:gDay ---20` bzw. `xsd:gDay ---31` für Anfang und Ende. Zur Verdeutlichung siehe Abbildung 4.9, in der zur besseren Übersichtlichkeit nicht alle Klassen und Relationen dargestellt sind.
Die Erweiterung der Time-Ontologie wurde so gestaltet, dass der Annotationsaufwand möglichst gering bleibt und der Annotator die Ontologie möglichst wenig kennen muss. Darauf wird weiter in Kapitel 5 eingegangen.

4.3 Die Mondontologie

Die Mondontologie dient lediglich dazu, die Informationen aus dem Web Service Evolutio Rodurago, dem Mondkalender, extrahierbar zu machen. Sie erfasst daher nur die Informationen, die der Web Service liefert. Diese sind: das Datum, das Sternzeichen, in dem sich der Mond gerade befindet, Mondauf- und -untergang und das Alter des Mondes innerhalb des Zyklus, woraus die Mondphase erschlossen werden

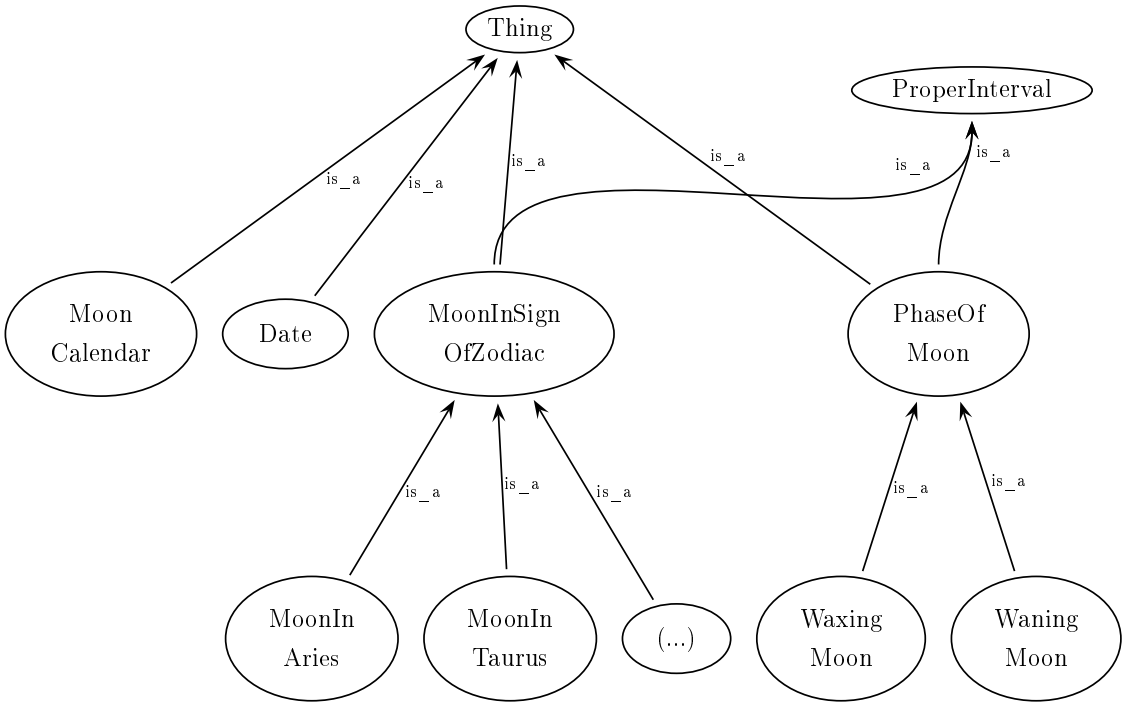


Abbildung 4.10: Mondontologie

kann. Da sich diese Begriffe nicht in Ober- und Unterklassen unterbringen lassen, ist die Ontologie eher flach und breit (siehe Abbildung 4.10).
Die Schlüsselklasse ist die Klasse `Date`, die mit allen anderen Klassen durch Eigenschaften in Verbindung steht. Damit werden dem Datum alle Informationen über

den Mond an diesem Datum zugewiesen. Daneben hat die Klasse die Datentyp-Eigenschaften `hasAgeOfMoonInXsdDecimal`, `hasMoonriseInXsdTime` und `hasMoonsetInXsdTime`. Erstere gibt als Dezimal-Datentyp das Mondalter an, wobei eine Zahl > 14 zunehmender Mond und eine Zahl ≤ 14 abnehmender Mond bedeutet. Die anderen beiden Eigenschaften geben als `xsd:time`-Datentyp die Uhrzeit von Mondauf- und Untergang an. Das Datum selbst wird mit der Datentyp-Eigenschaft `inXsdDate` als `xsd:Date` definiert (siehe Abbildung 4.11).

Dass es sich bei den Klassen `PhaseOfMoon` und `MoonInSignOfZodiac` um Inter-

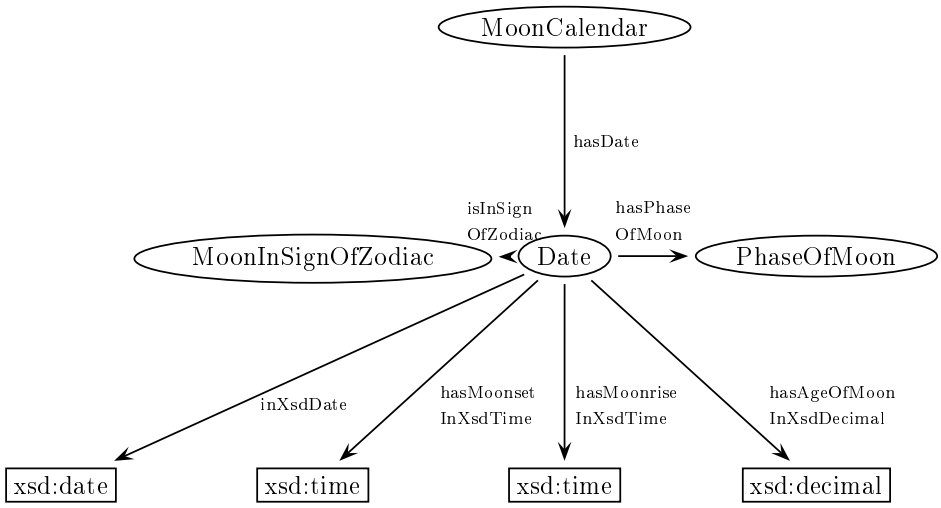


Abbildung 4.11: Die `Date`-Beschreibung

valle handelt, wird mit Hilfe von multipler Vererbung definiert. `PhaseOfMoon` und `MoonInSignOfZodiac` sind somit nicht nur Unterklassen von `MoonThing`, sondern zusätzlich von `ProperInterval` aus der Time-Ontologie.

4.4 OWL-S und WSDL

Die Web Ontology Language for Services, kurz OWL-S, ist eine OWL-Ontologie, die für die Beschreibung von Web Services entwickelt worden ist[26]. Als top-level-Ontologie enthält sie nur sehr allgemeine Konzepte, die domänenübergreifend auf alle Web Services anwendbar sein sollen. Um OWL-S für einen konkreten Web Service zu nutzen, muss die Ontologie mit den benötigten Individuen bestückt werden. Ziel ist es, den Web Service nicht nur für menschliche Nutzer zugänglich zu machen, sondern durch eine computerinterpretierbare Beschreibung auch Software-Agenten

[26] Vgl. Martin 2004

den Zugang zu ermöglichen. Die grundlegenden Konzepte von OWL-S spiegeln sich in den Kompetenzfragen wider, die sich im Zusammenhang mit Web Services stellen:

- Was leistet der Service? (Profile)
- Wie funktioniert der Service? (Model)
- Wie kann mit dem Service kommuniziert werden? (Grounding)

OWL-S besteht aus einer Service-Ontologie, die die Subontologien `Profile, Model` und `Grounding` einbindet. Der konkrete Web Service – hier der Mondkalender – wird als Instanz der Klasse `Service` gebildet. `Profile` bietet im Großen und Ganzen Informationen, die ein Agent braucht, um den Service im Netz zu finden, während `Model` und `Grounding` Informationen dazu liefern, wie der Agent den Service anwenden muss. Die oberste Ebene der Service-Ontologie mit ihren Unterontologien ist abgebildet in Abbildung 4.12[27].

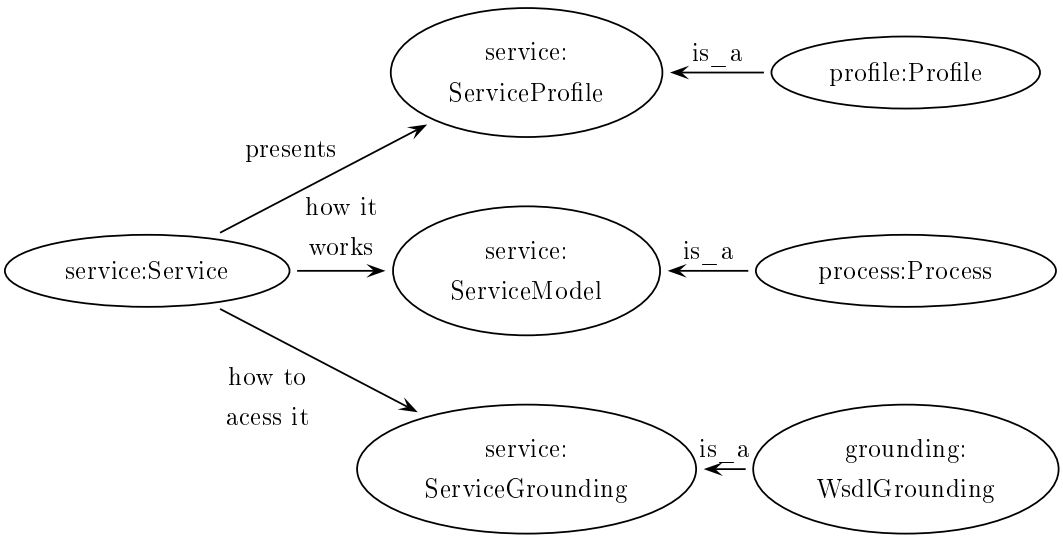


Abbildung 4.12: Die oberste Ebene der Service-Ontologie

4.4.1 Profile

Das Service-Profil dient in erster Linie dem automatisierten Auffinden des Web Services. Einerseits kann der Suchende seine Anforderungen an einen Dienst über `Profile` definieren und andererseits macht der Anbieter darüber die angebotene

[27] Vgl. Martin et al. 2004, http://www.daml.org/services/owl-s/1.1/overview/#tex2html3

Funktionalität publik. Im Semantic Web würde bei einer Anfrage quasi ein Wunsch-Service definiert, der dann mit den Services des Internets verglichen wird, wobei der Abgleich über einen Matchmarker erfolgt[28]. In der hiesigen Studie ist das Auffinden des Dienstes aber nicht implementiert.
Die Service-Eigenschaften `presents` und `presentedBy` sind inverse Relationen, die dem Zweck dienen, den Service mit dem Profil zu verbinden. Die Klasse `profile:Profile` enthält nun die Informationen, die den Service beschreibt. Es gibt dabei einige Eigenschaften, die nur für menschliche Leser gedacht sind wie `profile:serviceName`, mit der genau ein Name des Services, z.B. zu dessen Identifizierung, angegeben werden muss; es gibt `textDescription`, in der genau eine Kurzbeschreibung über den Service geliefert werden muss, und es gibt die Eigenschaft `contactInformation`, die angibt, wie man mit dem Serviceanbieter in Kontakt treten kann. Diese Eigenschaft hat keine Kardinalitätsbeschränkung im Gegensatz zu den anderen beiden.
Die funktionalen Komponenten des Web Services beschreiben die Eigenschaften `profile:hasInput, profile:hasOutput` und noch einige andere, die Voraussetzungen für den Service und Effekte des Services angeben, die für den Mondkalender aber nicht gebraucht wurden. Der Mondkalender benötigt zwei Eingabewerte: den Monat und das Jahr. Als Output gibt er jeden Tag eines Kalendermonats zurück.

```
1 <profile:Profile rdf:ID="MondkalenderProfile">
2     <profile:hasInput rdf:resource="#Monat"/>
3     <profile:hasInput rdf:resource="#Jahr"/>
4     <profile:hasOutput rdf:resource="#Kalendermonat"/>
5 </profile:Profile>
```

Listing 4.2: Input- und Outputbeschreibung der `profile`-Ontologie

Die beiden Eigenschaften verbinden die Profil-Ontologie mit der Prozess-Ontologie, wo Input und Output definiert werden. Die Ressourcen `#Monat` und `#Jahr` aus den Zeilen 2 und 3 werden also in den Prozess-Klassen `process:Input` bzw. `process:Output` genau beschrieben.
Um Aussagen über die Qualität und Zuverlässigkeit eines Web Services zu machen, gibt es noch einige Profil-Attribute, die hier aber nicht implementiert worden sind.

4.4.2 Model

Das Service-Model gibt an, wie der Web Service funktioniert, indem es ihn als einen Prozess beschreibt. Um die im Profil dargestellten Ergebnisse zu erzielen, muss der Benutzer diesen Prozess Schritt für Schritt ausführen. OWL-S unterscheidet zwischen atomaren und zusammengesetzten Prozessen. Ein atomarer Web Service ist

[28] Vgl. Hess, 2006, S. 3 und 4f

ein Online-Dienst, der nur eine einzige Anfrage entgegennimmt, seine Aufgaben ausführt und eine einzige Antwort zurückgibt, ohne dass weitere Interaktionen zwischen Klient und Service nötig werden. Auch in einem atomaren Prozess können beliebig viele Inputs eingegeben und beliebig viele Outputs generiert werden, allerdings werden sie mit nur einer Nachricht hin- und hergeschickt. Der Mondkalender ist beispielsweise ein atomarer Web Service mit einem atomaren Prozess. Er braucht zwei Eingaben – nämlich Monat und Jahr – und gibt eine Ausgabe zurück – den gesuchten Kalendermonat. Ein zusammengesetzter Prozess besteht aus mehreren atomaren Prozessen, hat also mehrere Stadien. Jede Nachricht, die der Klient dem Service schickt, führt ihn einen Schritt weiter im Prozessablauf. Ein Beispiel wäre der Kauf eines Buches bei amazon.de, der mehrere Interaktionen des Klienten, also mehrere Prozessschritte nötig macht.
Die Eigenschaften `service:describedBy` und `service:describes` verbinden den Service und das Model miteinander und sind inverse Eigenschaften. Auch im Model gibt es die Eigenschaften `process:hasInput` und `process:hasOutput`, die hier aber einen anderen Namensraum als die Profil-Eigenschaften haben und deshalb nicht mit ihnen zu verwechseln sind. Sie stellen eine Verbindung zu den Klassen `process:Input` und `process:Output` her, wo der Datentyp der Ein- und Ausgabe spezifiziert wird.
Für die Beschreibung des Mondkalenders sind diese Angaben bereits ausreichend. Komplexere Web Services bedürfen noch vieler weiterer Klassen und Eigenschaften[29], auf die hier aber nicht weiter eingegangen wird.

4.4.3 Grounding

Während `ServiceProfile` und `ServiceModel` rein abstrakte Repräsentationen des Dienstes sind, wird mit dem `ServiceGrounding` die konkrete technische Ebene des Web Services beschrieben – die abstrakte Beschreibung wird „geerdet“. Hier werden spezifische Details zu Nachrichtenformaten, Transportmechanismen, Protokollen, zur Adressierung etc. gegeben. Das `ServiceGrounding` gibt an, wie der abstrakt beschriebene Input und Output eines atomaren Prozesses in eine konkrete Nachricht umgesetzt und übermittelt werden kann. Dafür benützt OWL-S die Web Service Description Language (WSDL).
WSDL wurde unabhängig von Semantic Web-Technologien entwickelt und soll einen automatischen Zugriff auf Web Services ermöglichen. Mit WSDL stehen Informationen in maschinenlesbarer Form zur Verfügung. Allerdings wird der Service dabei nur auf syntaktischer Ebene beschrieben. Damit ist zwar das Interface, aber nicht die tatsächlich angebotene Funktionalität maschinenlesbar. Insofern stellt die syntaktische

[29] Vgl. Martin 2004, http://www.w3.org/Submission/OWL-S/#5

Service-Beschreibung von WSDL eine fruchbare Ergänzung zu der semantischen Beschreibung mit OWL-S dar. Die letzte WSDL-Version 2.0 ist eine W3C-Empfehlung vom 26.06.07, für die es aber noch keine OWL-S-Anpassung gibt. Deshalb wurde hier mit der WSDL- Vorgängerversion 1.1 gearbeitet, die das aktuelle OWL-S unterstützt. Für ein OWL-S/WSDL-Grounding ist die vollständige Spezifikation von beiden Sprachen erforderlich, da sie nicht die gleichen Konzepte abdecken. Vielmehr ergänzen sie sich auf komplementäre Weise: Während OWL-S die semantische Ausdruckstärke der Description Logic enthält, ist WSDL in der Lage, die abstrakte Repräsentation an einen konkreten Service zu binden. Wie in Abbildung 4.13[30] deutlich wird, gibt es einige Überlappungen beider Sprachen. Ein atomarer Prozess in

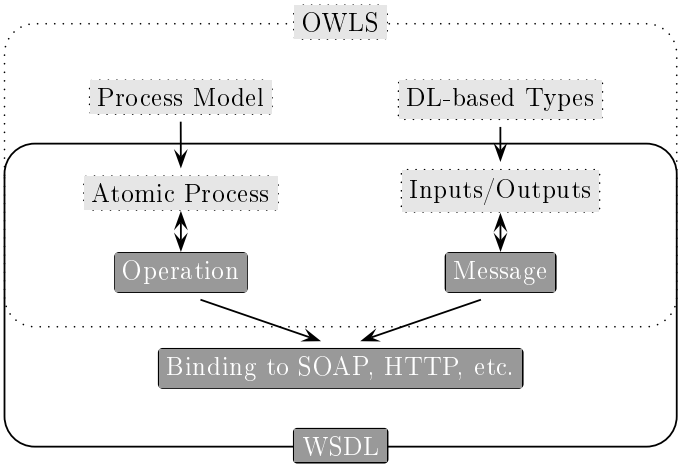


Abbildung 4.13: Das Zusammenwirken von OWL-S und WSDL

OWL-S ist eine Operation in WSDL, und ein OWL-S-Input/Output ist eine WSDL-Message. Das Grounding basiert nun auf diesen Überlappungen.
Bei der Implementierung müssen zuerst in WSDL eine Operation und die Messages definiert werden, bevor Korrespondenzen mit OWL-S zugewiesen werden können.
Im OWL-S-Grounding wird zunächst über die Eigenschaft `service:support` bzw. `service:supportedBy` eine Verbindung zwischen der Klasse `grounding:WsdlGrounding` und dem Service erstellt. Ferner gibt es über `hasAtomicProcess` eine Relation zu `grounding:WsdlAtomicProcess`. Da ein atomarer Prozess einer WSDL-Operation entspricht, ist diese Klasse der Elternknoten für die gesamte Abwicklung des Anfrage/Rückgabe-Prozesses. Hier wird zunächst eine Verbindung einerseits zu dem Model über `grounding:owlsProcess` und andererseits zu dem WSDL-Dokument über `grounding:wsdlDocument` aufgebaut, was die Schnittstellen-Rolle des OWL-S-Grounding deutlich macht.
OWL-S bildet den OWL-S-Input auf eine WSDL-Message ab. Im Quellcode sieht

[30] Vgl. Martin et al. 2004, http://www.daml.org/services/owl-s/1.1/overview/#tex2html3

das für den Eintrag *Monat* folgendermaßen aus:

```
<grounding:wsdlInput>
    <grounding:WsdlInputMessageMap>
        <grounding:owlsParameter rdf:resource="#MondMonat"/>
        <grounding:wsdlMessagePart rdf:datatype=
            "&xsd;#anyURI">&groundingWSDL;
            #month</grounding:wsdlMessagePart>
    </grounding:WsdlInputMessageMap>
</grounding:wsdlInput>
```

Listing 4.3: Mapping des OWL-S-Input auf eine WSDL-Message

Der Output wird ebenso behandelt wie der Input. Er ist hier nur sehr viel umfangreicher als der Input, weil der Inhalt des Mondkalenders mittels eines XSLT-Stylesheets als RDF-Dokument zurückgegeben wird. Da das Stylesheet nicht sehr groß ist, wurde es direkt als Transformationsstring in das `grounding:wsdlOutput`-Element geschrieben.
Das WSDL-Dokument definiert dann die Einträge konkret als `gMonth` und `gYear`:

```
<wsdl:message name="MondkalenderInputMessage">
  <wsdl:part name="month" type="s:gMonth" />
  <wsdl:part name="year" type="s:gYear" />
</wsdl:message>
```

Listing 4.4: Definition der Input-Datentypen

Stark verkürzt gesagt sorgt die WSDL-Operation dafür, dass die Message an die richtige Stelle des Webformulars geschrieben wird:

```
<wsdl:operation name="MondkalenderAnfrage">
    <http:operation location="/index.php?month=
        (month)&amp;year=(year)&amp;geodata=
        52.33%2C13.22%2C1&amp;site=details&amp;link=calendar"/>
    <wsdl:input>
        <http:urlReplacement />
    </wsdl:input>
    <wsdl:output>
        <mime:mimeXml part="Body" />
    </wsdl:output>
</wsdl:operation>
```

Listing 4.5: WSDL-Operation

Die URL des Web Services wird im WSDL-Element `Service` angegeben:

```
<wsdl:service name="MondKalenderService">
```

```
2     <wsdl:port name="MondkalenderHttpGet" binding
3         ="tns:MondkalenderHttpGetBinding">
4     <http:address location="http://www.rodurago.de" />
5     </wsdl:port>
6 </wsdl:service>
```

Listing 4.6: WSDL-Service

In WSDL werden also für jede Operation ein konkretes Nachrichtenformat und Details zu dem Übertragungsprotokoll spezifiziert. Dieser Vorgang wird auch als *binding* bezeichnet.

4.5 Resümee

Bei der Entwicklung und Bearbeitung der Ontologien wurde zweierlei deutlich: Ontologien sind nicht nur schwierig zu erstellen, sondern auch schwierig zu nutzen. Aufgrund der Komplexität der realen Welt ist es ausgesprochen schwer, eine sinnvolle Konzeptualisierung der Dinge zu erstellen. Die vielen zunächst diffus erscheinenden Möglichkeiten, die dem Entwickler bei der Erstellung einer Ontologie offen stehen, müssen sorgfältig gegeneinander abgewogen und ausprobiert werden, um eine gute Lösung zu finden. Generell versuchen Ontologien, Wissensgebiete möglichst eindeutig abzubilden. Je komplexer aber ein Wissensgebiet ist, umso größer ist die Gefahr, dass die Konzeptualisierung der Ontologie nicht mehr von allen Nutzern nachvollzogen wird, was die Arbeit mit dieser Ontologie erschwert und unnatürlich macht. Um diese Gefahr abzumildern sind übersichtliche Dokumentationen unumgänglich. Das Spannungsfeld von Anwenderkomfort und Übersichtlichkeit ist ein Balanceakt: Viele Klassen, die sämtliche Konstrukte aus den Webseiten unmittelbar darstellbar machen, werden leicht unübersichtlich und sind dadurch auch nicht mehr anwenderfreundlich. Bei wenig Klassen muss der Annotator, der die Ontologie nützt, gegebenenfalls mehr Kenntnisse über die Ontologie mitbringen, um die Objekte der Webseiten unterzubringen. Dafür ist die Menge der Klassen und Eigenschaften leichter zu durchschauen.
Das Erstellen und Bearbeiten einer funktionierenden, durchdachten Ontologie erfordert sehr viel Fachwissen und Erfahrung. Da es keine standardisierte Vorgehensweise für die Entwicklung einer Ontologie gibt, ist es wichtig, dass eine Ontologie über einen längeren Zeitraum entwickelt wird und viele Testphasen durchläuft. Eine enge Zusammenarbeit von Anwendern und Entwicklern für Rückmeldungen kann dabei nur hilfreich sein.

5 Die semantische Annotation der Seiten mit RDF

Mit der semantischen Annotation einer Webseite werden dem Dokument die Metainformationen zugefügt, die die Bedeutung der enthaltenen Informationen formalisiert wiedergeben. Diese Metainformationen erlauben es Computern, die Semantik der natürlichen Sprache zu verarbeiten. Im Prinzip bildet die Annotation die Schnittstelle zwischen der Ontologie und der Webseite. Die Tripel, die über den Inhalt der Seiten gebildet werden, stellen die Instanzen der Ontologie und ihre Eigenschaften dar.
Bei der Annotation bleibt zu bedenken, dass sich bei freiem, natürlich sprachlichem Text immer die Frage stellt, welche Textteile wichtig sind und annotiert werden müssen und welche so sehr ins Detail gehen, dass sie keine Relevanz mehr besitzen. Bei der semantischen Annotation spielt das persönliche Urteil des Annotators und die Anwendung, für die die Annotation gedacht ist, somit eine wichtige Rolle. Bei stärker strukturierten Texten ist die relevante Information durch Überschriften, Spiegelstriche, Tabellen etc. deutlicher hervorgehoben, was die Annotation sehr viel leichter macht.
Es gibt unterschiedliche Möglichkeiten, die Annotierung abzulegen[31]. Zum einen kann man die Metadaten direkt in die Webseite einbetten. Damit die Daten nicht vom Web Browser dargestellt werden, müssen sie entweder in einem HTML-Kommentar `<!-...->` auskommentiert oder direkt in den Kopf der Datei geschrieben werden. Letztere Methode hat allerdings den Nachteil, dass das HTML- oder XML-Dokument dann nicht mehr valide ist. Es gibt ferner zwei HTML-Elemente, `<object>` und `<script>`, mit denen man nicht-HTML Medien einbinden kann. Bei der vom W3C empfohlenen Variante[32] wird lediglich ein Link auf die Webseite geleget, der auf das RDF-Dokument mit der Annotation verweist. Die Metadaten sind in diesem Fall in einer eigenen Datei separat abgelegt.
Weil ein Agent nur die Annotation braucht, um die gesuchten Informationen zu extrahieren und die Webseiten einzig der Lesbarkeit für den Menschen dienen, sind in dieser Studie die HTML-Seiten einfachheitshalber überhaupt nicht mit den RDF-Dokumenten verbunden, sondern werden direkt dem Agenten übergeben. Ansonsten

[31] Vgl. Reif, 2006, S. 406ff
[32] Vgl. W3C-FAQ, http://www.w3.org/RDF/FAQ.html#How

hätte die Annotation extern verlinkt werden müssen, weil keine Zugriffsrechte auf die Internetseiten bestehen.

5.1 Manuelle Annotation der Gartenseiten

Die statischen Webseiten wurden rein manuell annotiert. Für jede Internetseite existiert ein korrespondierendes RDF-Dokument, das die semantisch aufbereitete Information der Seite soweit enthält, wie es die Anwendung erfordert. Eine vollständige Annotation ist im Rahmen der Studie nicht erfolgt.
Die Seiten binden verschiedene Namensräume ein, abhängig davon, welche Elemente in der Annotation verwendet werden. Das sind hier der `xsd-`, `rdf-` und `owl`-Namensraum, sowie der Namensraum der Gartenontologie. Wird eine Ressource ohne Namensraum-Präfix verwendet, bildet das System einen Default-Namensraum, der, wenn nicht anders angegeben, dem lokalen Namensraum der RDF-Datei entspricht. Jede annotierte Seite dieser Studie hat dadurch seinen eigenen Namensraum. Alternativ könnte man die Seiten mit den einzelnen Gemüsearten in einem großen Namensraum zusammenfassen. Das hätte den Vorteil, dass gemeinsam genutzte Ressourcen nur einmal definiert werden müssten. Wenn Kopfsalat und Tomate beispielsweise jeweils Ende Februar gesät werden müssen, so wäre die einmalige Definition der Ressource *Ende Februar* ausreichend, anstatt sie wie hier sowohl in dem RDF-Dokument von Kopfsalat als auch in dem RDF-Dokument von Tomate zu definieren. Bei einer größeren Anzahl von Gemüsesorten käme es aber sehr leicht zu Namenskollisionen, da der Namensraum durch seine Größe kaum mehr zu überblicken wäre, was den Annotierungsprozess unnötig erschweren würde. Aus diesem Grund wurden die Redundanzen, die durch die kleinen Namensräume zustande kommen, in Kauf genommen.

5.1.1 Die Seite *Gemüsegarten*

Diese Seite enthält keine inhaltliche Information für den Agenten, sondern lediglich die Links, die auf die annotierten Seiten über Kopfsalat, Tomate und Buschbohne verweisen.
Für die Annotation muss zunächst die Gartenontologie importiert werden. Dafür gibt es das OWL-Element `import`. Die Verweise auf die relevanten Seiten sind folgendermaßen annotiert:

```
1    <rdf:Description rdf:about="&gemuese;/gemuesegarten.htm">
2        <garden:Vegetable rdf:parseType="Collection">
3            <rdf:Description rdf:about="&tom;Tomate"/>
4            <rdf:Description rdf:about="&ks;Kopfsalat"/>
```

```
5             <rdf:Description rdf:about="&bb;Buschbohne"/>
6         </garden:Vegetable>
7     </rdf:Description>
```

Listing 5.1: Annotation von Verweisen auf andere Seiten

Die erste Zeile nennt die Ressource, die beschrieben werden soll, das Subjekt des Tripels also, das hier allgemein die Seite *Gemüsegarten* ist. `garden:Vegetable` in Zeile 2 enthält die Information, um welche Art von Ressource es sich bei den Zielen handelt – um irgendeine Art Gemüse nämlich. Das Attribut `rdf:parseType=''Collection''` drückt aus, dass alle eingebetteten Elemente in einer Listenstruktur zusammengefasst sind. Diese Elemente (Zeilen 3, 4 und 5) sind die Ressourcen, von denen der Agent seine Informationen über die gesuchten Gemüsesorten erhält. Die Entity `&<...>;` gibt den Pfad zu der annotierten Gemüseseite an, und das unmittelbar folgende Gemüse verweist auf die Stelle des Dokuments, an der es definiert ist.
Die Verweise könnten auch mit dem `rdfs:seeAlso`-Element annotiert werden. Der Verweis auf die annotierte Tomatenseite sähe dann aus wie in Listing 5.2:

```
1     <rdf:Description rdf:about="Tomate">
2         <rdfs:seeAlso rdf:resource="&tom;Tomate"/>
3     </rdf:Description>
```

Listing 5.2: Alternative Annotation von Verweisen mit `SeeAlso`

Mit `rdfs:seeAlso` wird eine Relation zu einer Ressource erstellt, die weitere Informationen zu der Ressource des Definitionsbereichs liefert. Die Annotation mit `seeAlso` hat den Nachteil, dass der Agent mit dieser Relation noch nicht weiß, dass, wie in diesem Beispiel, eine Tomate ein Gemüse ist. Das erfährt er erst auf der annotierten Tomatenseite. Wenn nun Verweise über unterschiedliche Objekte wie Gemüse, Boden, Mischkultur etc. annotiert sind und der Agent nach Informationen zu einem Gemüse sucht, würde er alle Verweise verfolgen müssen, um zu erfahren, welcher Verweis zu Gemüseinformationen verhilft.

5.1.2 Die Seite *Kopfsalat*

Da die Gemüseseiten sehr ähnlich aufgebaut sind, soll es hier genügen, beispielhaft die Annotation der Kopfsalatseite zu erklären. Der Agent soll aus den Gemüseseiten die Informationen zu Standort, Bodenbeschaffenheit und Saatzeit extrahieren. Ist diese Information einmal für eine Seite annotiert, bedeutet es keinen großen Aufwand mehr, die Annotation für andere Gemüsesorten zu erstellen, abhängig natürlich von der Anzahl der zu annotierenden Gemüsesorten.
Wie bei der Gemüseseite werden zunächst die Namensräume deklariert und die Gartenontologie importiert. Für die eigentliche Beschreibung des Kopfsalates muss eine

Ressource `Kopfsalat` als Instanz der Gartenontologie-Klasse `ButterheadLettuce` deklariert werden. Diese Relation wird durch das RDF-Element `type` erzeugt (siehe Listing 5.3, Zeile 2). Wenn das Individuum `Kopfsalat` nun existiert, können ihm Eigenschaften aus der Gartenontologie wie `needsGround`, `needsClimate`, `isSeededIndoor` etc. zugewiesen werden. Diese Eigenschaften müssen alle den Definitionsbereich `Kopfsalat` erlauben. Da die Eigenschaften `needsGround` (Zeile 3) und `needsClimate` (Zeile 10) mehrere Werte haben, werden diese in einem Bag definiert. Der etwas gekürzte Quellcode für die Beschreibung `Kopfsalat` sieht folgendermaßen aus:

```
1   <rdf:Description rdf:about="Kopfsalat">
2       <rdf:type rdf:resource="&garden;ButterheadLettuce"/>
3       <garden:needsGround>
4           <rdf:Bag>
5               <rdf:_1 rdf:resource="TiefgruendigerBoden"/>
6               <rdf:_2 rdf:resource="DurchlaessigerBoden"/>
7               <rdf:_3 rdf:resource="HumoserBoden"/>
8           </rdf:Bag>
9       </garden:needsGround>
10      <garden:needsClimate>
11          <rdf:Bag>
12              <rdf:_1 rdf:resource="SonnigesKlima"/>
13              <rdf:_2 rdf:resource="WindgeschuetztesKlima"/>
14          </rdf:Bag>
15      </garden:needsClimate>
16      <garden:isSeededOutdoor rdf:resource=
17          "SaatzeitSalatDraussen"/>
18      <garden:isSeededOutdoor rdf:resource="Gemuesegarten"/>
19      <garden:isSeededIndoor (...)>
20  </rdf:Description>
```

Listing 5.3: Annotation von Kopfsalat

Alle hier verwendeten Ausdrücke wie `DurchlaessigerBoden`, `SonnigesKlima`, `SaatzeitSalatDrinnen` etc. müssen mit `rdf:type` einer Klasse der Gartenontologie zugewiesen werden, damit sie auch für Maschinen als Mitglieder einer bestimmten Klassen erkenntlich sind. Die RDF-Ausdrücke dafür haben die Form:

```
1   <rdf:Description rdf:about="RESSOURCE">
2       <rdf:type rdf:resource="&garden;KLASSE"/>
3   </rdf:Description>
```

Listing 5.4: Zuweisung von RDF-Ressourcen zu einer Ontologieklasse

Da die Beschreibung der Saatzeit einige Probleme in sich birgt, soll darauf am Beispiel `SaatzeitSalatDraussen` noch etwas genauer eingegangen werden. Zunächst der Code:

```
1   <rdf:Description rdf:about="SaatzeitSalatDraussen">
2       <rdf:type rdf:resource="&garden;SeedingInterval"/>
3       <time:hasBeginning rdf:resource="April"/>
4       <time:hasEnd rdf:resource ="EndeJuli"/>
5   </rdf:Description>
6
7   <rdf:Description rdf:about="EndeJuli">
8       <rdf:type rdf:resource=
9           "&garden;InstantThirdPartOfMonth31"/>
10      <time:inDateTime rdf:resource="&garden;July"/>
11  </rdf:Description>
12
13  <rdf:Description rdf:about="April">
14      <rdf:type rdf:resource=
15          "&garden;InstantFirstPartOfMonth"/>
16      <!--nicht sauber annotiert!-->
17      <time:inDateTime rdf:resource="&garden;April"/>
18  </rdf:Description>
```

Listing 5.5: Annotation der Saatzeit

Die Webseite gibt an, dass man Salat von April bis Ende Juli draußen säen kann. Nachdem die Ressource `SaatzeitSalatDraussen` als ein `SeedingInterval` deklariert worden ist (Zeile 2), werden ihr mit den Eigenschaften `hasBeginning` und `hasEnd` Anfangs- und Endzeitpunkt zugewiesen (Zeilen 3 und 4), was intuitiv noch leicht zu verstehen ist. Um aber dann den Zeitpunkt `EndeJuli` zu beschreiben, muss der Annotator die Ontologie recht gut kennen. Er muss nicht nur wissen, dass er den Zeitpunkt der Klasse `InstantThirdPartOfMonth31` (Zeile 9) zuweisen muss, weil es sich bei dem Zeitpunkt um den dritten Teil eines Monats mit 31 Tagen handelt, sondern auch, dass er den Monat Juli direkt an eine Date-Time-Description übergeben muss (Zeile 10), weil sonst die Information verloren geht, um welchen konkreten Monat es sich handelt (siehe zur Erinnerung Abbildung 4.8, Seite 37). Die Verbindung zur Date-Time-Description hätte verhindert werden können, wenn jeder Monat mit seinen drei Teilen sowohl in der Klasse `Instant` als auch in der Klasse `Interval` aufgelistet wäre. Dann müsste der Annotator seinen Zeitpunkt nur der richtigen Klasse zuordnen. Es ergäbe sich allerdings ein anderes Problem: Die Saatzeiten sind auf den Webseiten fast immer mit den Monatsdritteln ausgedrückt. Die einzige Ausnahme ist der Zeitpunkt `April` auf der Kopfsalatseite. Dieser Zeitpunkt kann mit

dem momentanen Stand der Ontologie nicht korrekt wiedergegeben werden, weil die Ontologie keine Klasse wie `InstantApril` hat. Das Problem ist, dass die Sprache zu facettenreich ist, um sämtliche Möglichkeiten abzudecken. Würde man noch alle Monate als Zeitpunkte aufnehmen und vielleicht auch noch die Monatshälften, hätte man bereits 21 Klassen, wenn man die unterschiedlichen Monatslängen mitberücksichtigt, was die Ontologie immer unübersichtlicher machen würde.
Eine bessere Alternative wäre möglicherweise, einen Zeitpunkt wie `EndeJuli` oder `April` als Instanz der Klasse `Instant` zu deklarieren und nicht als Klasse; dann wäre zumindest die Vielfalt der Sprache kein Problem mehr. Allerdings müsste der Annotator dann noch mehr Kenntnisse über die Time-Ontologie und deren Möglichkeiten mitbringen. Dann müsste er nämlich seinem instanziierten Zeitpunkt noch zusätzlich die Information geben, dass der zu beschreibende Zeitpunkt irgendwo innerhalb eines speziellen Intervalls liegt, dass er für diese Verbindung die Eigenschaft `inside` bräuchte und dass Anfangs- und Endzeitpunkt des Intervalls anderweitig definiert sind.
Eine weitere Überlegung wäre, ob man nicht den Eigenschaften `hasBeginning` und `hasEnd` eine Ressource übermittelt, die direkt eine Datumsangabe enthält – in der Time-Ontologie eine `DateTimeDescription`. Das wäre auf jeden Fall das einfachste und könnte intuitiv am leichtesten nachvollzogen werden. Die vorausgesetzten Kenntnisse über die Ontologie würden dadurch auf ein Minimum reduziert. Mit der Konzeption der Time-Ontologie hätte das aber nicht mehr viel zu tun. Auch wäre der Inhalt der Webseite nicht exakt wiederzugeben, weil man nicht mehr modellieren könnte, dass es sich bei einem Ausdruck wie *Ende Juli* um ein Intervall handelt. Auch müsste sich jeder Annotator selbst entscheiden, mit welchen Kalendertagen er eine solche Zeitangabe fixiert. Bei mehreren Annotatoren käme es zwangsweise zu Diskrepanzen.
In der hiesigen Implementierung wurde versucht, einen Mittelweg zwischen den Vor- und Nachteilen zu finden: Dem Annotator sollten nicht allzu viele Ontologie-Kenntnisse zugemutet werden, die Anzahl der Klassen sollte sich noch im Rahmen halten und die korrekte Wiedergabe des Seiteninhalts sollte einigermaßen gewährleistet sein. Bisher sind diese Kriterien nur annäherungsweise erreicht. Verbesserungen stehen noch aus.

5.1.3 Die Seite *Kleingärtnerin*

Von dieser Seite wurde die Tabelle annotiert, die die guten und schlechten Nachbarn einer Gemüseart angibt. Die tabellarische Auflistung ist in dem RDF-Dokument in großen Containern wiedergegeben, die die Nachbarn enthalten. Als Beispiel sei die Annotation der Nachbarschaftsverhältnisse von Kopfsalat in gekürzter Ausführung

gegeben:

```
<rdf:Description rdf:about="Kopfsalat">
    <garden:inGoodNeighbourhoodWith>
        <rdf:Bag rdf:ID="GoodNeighbourSalad">
            <rdf:_1 rdf:resource="Bohne"/>
            <rdf:_2 rdf:resource="Kohl"/>
            (...)
        </rdf:Bag>
    </garden:inGoodNeighbourhoodWith>
    <garden:inBadNeighbourhoodWith>
        <rdf:Bag rdf:ID="BadNeighbourSalad">
            <rdf:_1 rdf:resource="Kresse"/>
            <rdf:_2 rdf:resource="Petersilie"/>
            (...)
        </rdf:Bag>
    </garden:inBadNeighbourhoodWith>
</rdf:Description>
```

Listing 5.6: Annotation von Nachbarschaftsverhältnissen bei Kopfsalat

5.2 Automatische Annotation mit XSL

Bei der Annotation der Mondkalenderseiten musste anders vorgegangen werden als bei den übrigen statischen Webseiten. Bei jedem Aufruf des Mondkalenders wird eine neue HTML-Seite generiert, die die Informationen des gewünschten Kalendermonats liefert. Die zurückgegebenen Seiten sind im Prinzip jedes Mal gleich, nur dass die Werte bei jedem Ergebnis neu eingetragen werden. Daher bietet es sich an, die Annotation mit Hilfe eines XSLT-Stylesheets zu erstellen. Der Quellbaum ist dabei die vom Web Service generierte HTML-Seite; der Ergebnisbaum ist das gewünschte RDF-Dokument. Das XSLT-Dokument definiert die Schablone des RDF-Dokuments und setzt dort Platzhalter ein, wo im Ergebnisbaum die Werte eingetragen werden sollen. Da jeder Kalendertag gleich codiert ist, besteht das Stylesheet weitestgehend aus der Schablone eines einzigen Kalendertages und wiederholt sich bis alle Tage des Monats abgearbeitet sind.

Da das Stylesheet in seinem Zeilenumfang relativ klein ist, steht es nicht in einer eigenen Datei, sondern ist direkt in dem `<grounding:wsdlOutput>`-Element der OWL-S-Datei als Transformationsstring aufgeführt. Die Implementierung des Stylesheets findet sich auf der beiliegenden CD-Rom.

6 Semantic Web-Werkzeuge

Für die Bearbeitung der unterschiedlichen Semantic Web-Technologien gibt es inzwischen eine ganze Reihe von Werkzeugen auf dem Markt, die den Prozess der semantischen Annotation und das Erstellen von Ontologien erleichtern sollen. Im Allgemeinen verfolgen die Werkzeuge das Ziel, auch Benutzern mit möglichst wenig RDF- oder OWL-Kenntnissen das Arbeiten mit Semantic Web-Technologien zu ermöglichen, indem sich beispielsweise das Werkzeug um die Syntax kümmert, die sehr häufig gebrauchten URIs nicht eingegeben werden müssen etc. Einige dieser Werkzeuge wurden in dieser Studie ausprobiert und sollen in diesem Kapitel mit ihren Vor- und Nachteilen vorgestellt werden.

6.1 Protégé

Die hier gegebene Beschreibung bezieht sich auf Protégé 3.2.1 von Dezember 2006. Seit dem 29.06.07 ist zusätzlich eine Beta-Version Protégé 3.3 und eine Alpha-Version Protégé 4.0 erhältlich.
Protégé[33] ist eine frei erhältliche Java-Applikation, die bereits in den 80er Jahren am Institut für Medizinische Informatik an der Stanford University entwickelt worden ist. Ursprünglich wurde Protégé zur Entwicklung von medizinischen Wissensdatenbanken konzipiert, ist inzwischen aber mit vielen Plugins, Erweiterungen und einer großen Benutzergemeinschaft zu einem der bekanntesten Ontologie-Editoren herangewachsen. Protégé bietet zwei Arten der Wissensmodellierung an. Zum einen gibt es Protégé-frames, das bestimmtes Domänenwissen in einer hierarchischen Struktur von Konzepten, Slots und Instanzen abbildet. Bei Protégé-OWL kommen im Gegensatz dazu noch logische Mechanismen dazu, durch die implizites Wissen aus dem modellierten Wissensgebiet erschlossen werden kann.
Die graphische Oberfläche von Protégé stellt die Ontologie sehr übersichtlich dar (siehe Abbildung 6.1). Auch große Datenmengen können problemlos verwaltet und bearbeitet werden. Es gibt ausführliche und gut verständliche Tutorials über Protégé[34], die den Einstieg ungemein erleichtern.
Protégé bietet mehrere graphische Oberflächen an, über die die Ontologie bearbeitet

[33] Vgl. Protégé-Homepage: http://protege.stanford.edu/
[34] Vgl. Protégé-Tutorial: http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf

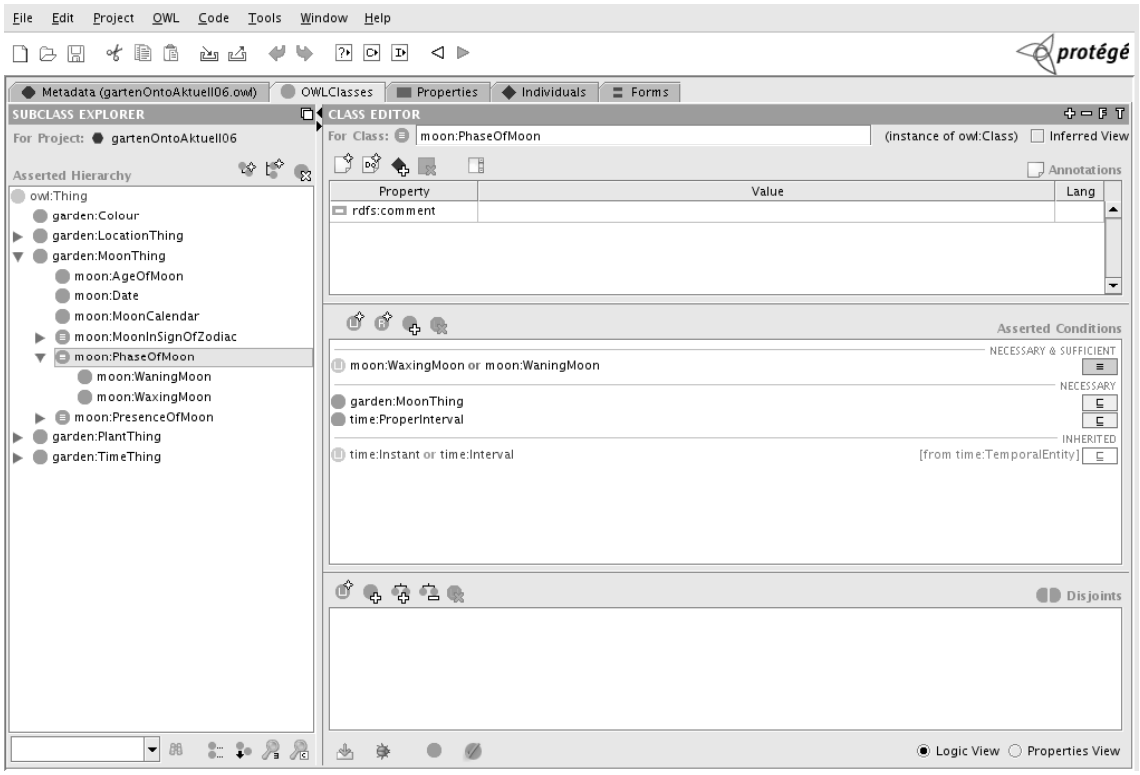


Abbildung 6.1: Die graphische Oberfläche von Protégé

wird. Es gibt Tabs für die Erstellung und Bearbeitung von Klassen, Eigenschaften, und Instanzen, für SPARQL-Anfragen, für die graphische Visualisierung der Ontologie und noch viele mehr. Zudem gibt es unzählige Fenster für unterschiedlichste Anwendungen wie: viele Klassen auf einmal erstellen, viele Klassen disjunkt zueinander machen, viele Beschränkungen auf einmal definieren etc. Die Bearbeitung über solche Fenster ist einerseits natürlich sehr einfach, andererseits gibt es zu wenig Möglichkeiten, diese Fenster mit Tastenkombinationen zu bedienen, was unnötig viel Klickerei mit der Maus erfordert. Der OWL-Code wird automatisch generiert, kann aber nicht vom Anwender direkt bearbeitet werden, was beim Debuggen durchaus von Nachteil ist.

Protégé 3.2.1 versucht möglichst natürlichsprachlich mit OWL-Ausdrücken umzugehen. Teilweise ist das etwas irritierend. Z.B. unterscheidet Protégé zwischen Beschränkungen, die *Necessary* oder *Necessary and Sufficient*, sind. Mit *Necessary and Sufficient* soll dabei ausgedrückt werden, dass mit diesen Beschränkungen eine Äquivalenzklasse gebildet wird. Wenn ein Individuum also Element der Klasse ist, dann muss es die Bedingung erfüllen und umgekehrt muss es, wenn es die Bedingung erfüllt Element der Klasse sein. Individuen die eine *Necessary*-Beschränkung haben, bilden dagegen eine Oberklasse. Ein Individuum ist Element der Klasse, wenn es die Bedingung erfüllt. Umgekehrt heißt das aber nicht, dass jedes Element, das die Bedingung erfüllt auch Element der Klasse ist.

Die Protégé-Versionen unter 4.0 haben keinen eingebauten Reasoner. Da die Version 4.0 alpha noch deutlich zu schlecht funktioniert (Stand 05.07), ist man darauf angewiesen einen Reasoner, z.B. Pellet[35], extern zu benützen. Der Reasoner kommuniziert mit Protégé über die Sprache DIG, die aber veraltet ist und verschiedene Datentypen wie `xsd:gDay, xsd:gMonth` oder auch einige OWL-Elemente wie das Komplement nicht erkennt. Der Reasoner mit Protégé ist für die hier bearbeitete Ontologie deshalb nicht nutzbar. Um die Konsistenz der Ontologie zu überprüfen, musste auf andere Werkzeuge zurückgegriffen werden.
Einige Probleme hat es auch beim Abspeichern gegeben. Die Werkzeuge haben unterschiedliche Konventionen den Quellcode abzuspeichern. Nachdem die hier verwendete Gartenontologie beispielsweise mit dem SWOOP-Editor (siehe unten) bearbeitet und abgespeichert wurde, konnte Protégé die Datei zwar problemlos öffnen und weiter bearbeiten, nach dem Abspeichern mit Protégé kam es dann allerdings zu gravierenden Fehlern im Quellcode, wodurch die Ontologie von den Werkzeugen nicht mehr dargestellt werden konnte. Das Problem waren leere Äquivalenzklassen, die in die Ontologie geraten waren. Der so entstandene Schaden war sehr schwer zu lokalisieren, weil das Dokument weiterhin valide war und Validatoren[36] keine Fehlermeldungen angaben. Der Reasoner Pellet konnte den Fehler zwar feststellen, allerdings gab er nicht an, in welcher Zeile der Fehler lag.
Es empfiehlt sich, den Default-Namensraum `xmlns` und das `xml:base`-Attribut in den Ontologien mit dem gleichen Wert anzugeben. Das `xml:base`-Attribut ist die Basis-URL, von der aus alle relativen Links in dem Dokument ausgewertet werden. Diese Angaben sind zwar nach der W3C-Empfehlung nicht obligatorisch. Allerdings kann es ansonsten bei einigen Werkzeugen wie auch dem Protégé immer wieder zu unberechenbaren Fehlern kommen: Mal wird die Klassenhierarchie gänzlich falsch dargestellt, mal gibt es eine Fehlermeldung, dass das Dokument nicht valide sei und wieder ein anderes Mal funktioniert das Programm tadellos.
Eigenschaften und Klassen müssen bei Protégé fertig angelegt sein, bevor Instanzen eingegeben werden können, da es zu Datenverlusten kommt, wenn die Klasse oder Eigenschaft einer Instanz noch einmal verändert wird. Dadurch kann man, bevor die Ontologie nicht fertig ist, keine Anfragen an die Ontologie stellen, um zu testen, ob die Ergebnisse richtig sind. Besonders für Anfänger ist das ein großer Nachteil.
Da mit Protégé auch Instanzen erstellt werden können, eigent sich das Programm im Prinzip auch zur Annotierung von Webseiten. Die Instanzen werden aber nicht in einer separaten Datei abgelegt, sondern in den Quellcode der Ontologie integriert. Eine Trennung von Ontologie und Annotation ist damit also nicht möglich.

[35] Vgl. Pellet-Homepage: http://pellet.owldl.com/
[36] Vgl. W3C Validation Service: http://www.w3.org/RDF/Validator/ARPServlet und WonderWeb OWL Ontology Validator: http://www.mygrid.org.uk/OWL/Validator

6.2 SWOOP

SWOOP 2.2.1[37] ist ein Ontologie-Editor, der von seinem äußeren Erscheinungsbild einem Web Browser sehr ähnlich ist, um den Benutzern eine vertraute graphische Oberfläche zu bieten. Die Ontologie kann geladen werden, indem die Adresse wie bei einem Web Browser in eine Adresszeile eingegeben wird, es gibt Lesezeichen, in denen Ontologien gespeichert werden können, es gibt eine Vor- und Zurück-Schaltfläche, mit der die letzte und vorherige Einstellung wieder aufgerufen werden kann. Zudem gibt es links eine schmale Spalte, die mit einer Navigationsleiste vergleichbar ist, in der die geöffneten Ontologien und der Ontologiebaum angezeigt werden. Rechts daneben befindet sich das große Fenster, in dem die Klassen im Detail mit allen Eigenschaften dargestellt werden. (siehe Abbildung 6.2) Neben dem abgebildeten

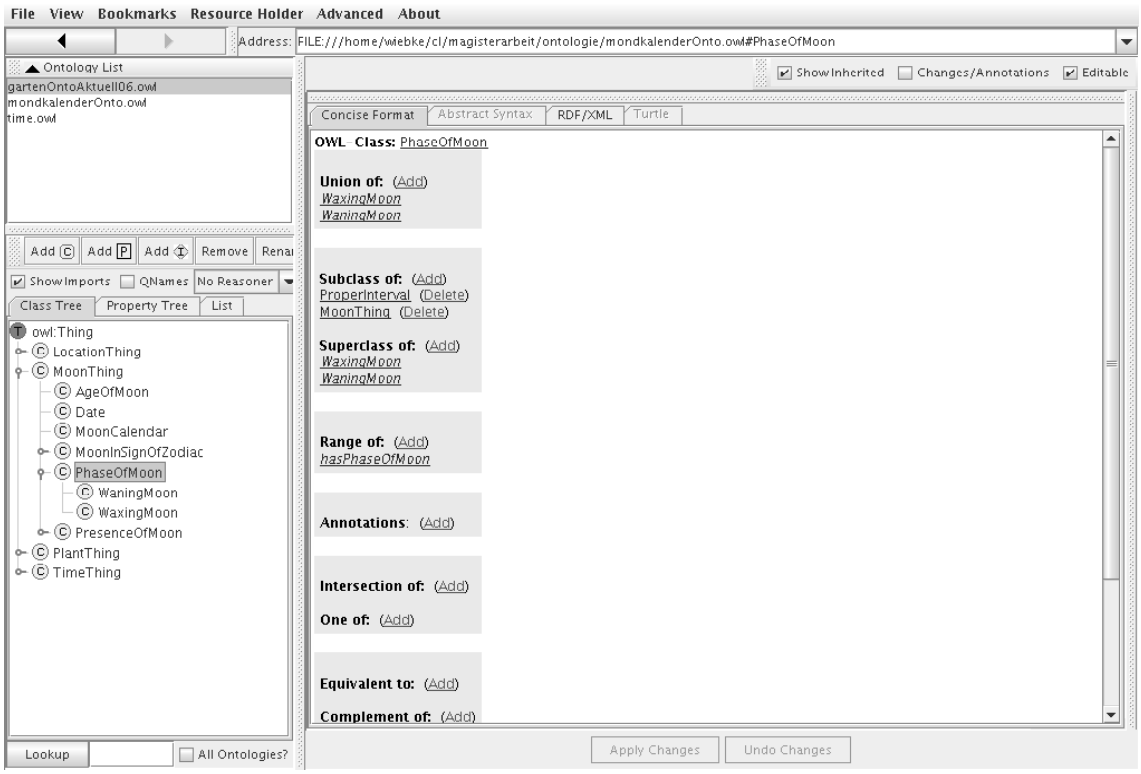


Abbildung 6.2: Die graphische Oberfläche von SWOOP

Concise Format, in dem Klassen, Eigenschaften und Instanzen bearbeitet werden können, hat man die Möglichkeit sich die Klasse direkt im Quellcode anzusehen und zu editieren.

Bei SWOOP ist der Reasoner Pellet direkt integriert, was die Kommunikation über DIG überflüssig macht.

Auch SWOOP hat einige „Schönheitsfehler“, die die Ontologie-Bearbeitung erschweren. So werden Eigenschaften nur bei der Klasse dargestellt, die diese Eigenschaften

[37] Vgl. Kalyanpur 2005; SWOOP-Homepage bei Google: http://code.google.com/p/swoop/

zugewiesen bekommt. Bei allen Unterklassen, die diese Eigenschaft erben, ist sie aber nicht sichtbar. Das hat bei der Bearbeitung von Beschränkungen oder bei der Instanziierung von Individuen den Nachteil, dass man erst in den Oberklassen nachsehen muss, ob die entsprechende Eigenschft überhaupt für die bearbeitete Klasse existiert.
Klassen, Eigenschaften und Individuen werden bearbeitet, indem sie aus einem Auswahlmenü ausgesucht werden. Das hat den Nachteil, dass man mit dem Größerwerden der Ontologie immer längere Auswahllisten hat. Bei der Gartenontologie mit gut 150 Klassen ist das bereits mühsam.
Größte Vorsicht ist auch hier beim Abspeichern geboten. In SWOOP kann man mehrere Ontologien auf einmal laden, was sehr praktisch ist. Die Ontologie, die gerade angezeigt wird und bearbeitet werden kann, ist dabei in dem Fenster *Ontology List* farblich hervorgehoben. Während alle geladenen Ontologien beliebig bearbeitet werden können, speichert SWOOP aber immer unter dem zuletzt geladenen Ontologie-Namen ab. Wenn man nun eine nicht zuletzt geladene Ontologie bearbeitet hat und abspeichert, wird die zuletzt geladene Ontologie unwiederbringlich überschrieben. Unter Umständen merkt man das erst beim nächsten Laden. Es lohnt sich also immer, für aktuelle Sicherungskopien zu sorgen.
SWOOP enthält einen integrierten XML-Editor, in den man alternativ direkt den Quellcode in XML eingeben kann. Einige Bugs sorgen hier noch für Überraschungen. Versucht man, eine importierte Klasse zu verändern, was grundsätzlich nicht möglich ist, so lehnt SWOOP nicht nur die Veränderung ab, sondern löscht zusätzlich noch einige vorherige Bearbeitungen. Wenn nicht gleich abgespeichert wird, lässt sich die vorherige Bearbeitung durch ein Reload retten, wenn man ihr Fehlen bemerkt hat. Auch dieser Editor speichert die Individuen nicht in einer separaten Datei ab, weshalb SWOOP wie auch Protégé nur bedingt für die Annotierung von Webseiten geeignet ist.

6.3 SMORE

SMORE[38] ist das Vorgängerprodukt von SWOOP und diesem in Bezug auf die Gestaltung der graphischen Oberfläche sehr ähnlich (siehe Abbildung 6.3). SMORE ist ein Editor, der primär dazu konzipiert ist, Webseiten simultan mit deren RDF-Markup zu erstellen, der aber auch das Erstellen und Bearbeiten von Ontologien ermöglicht. Das Werkzeug ist explizit für Anwender mit limitierten OWL-Kenntnissen gedacht. SMORE enthält einen HTML-Editor und einen Web Browser, mit dem man via Drag and Drop Textteile der HTML-Seite der Ontologie zuweisen kann. Wie bei

[38] Vgl. SMORE-Homepage: http://www.mindswap.org/2005/SMORE/

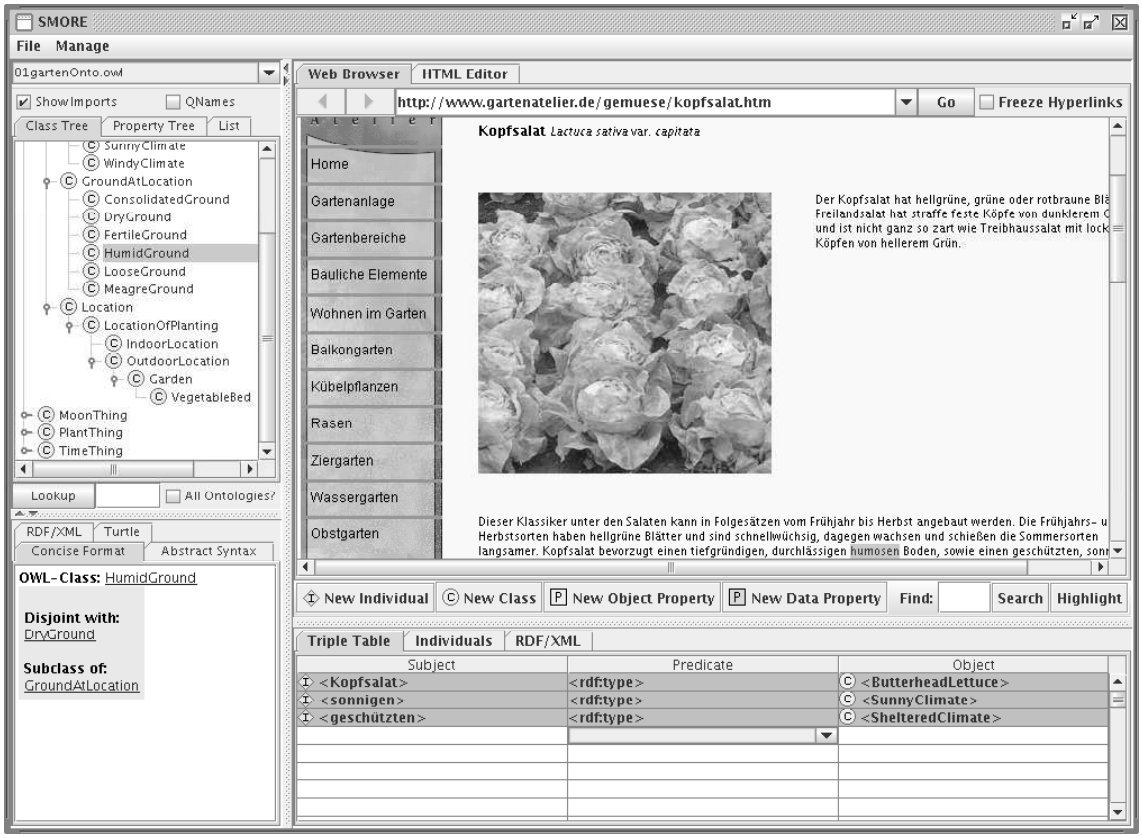


Abbildung 6.3: Die graphische Oberfläche von SMORE

SWOOP ist auch hier angenehm, dass man den XML/RDF-Code direkt ansehen, wenn auch leider nicht bearbeiten kann.

Auch wenn es sich bei dem hier getesteten Werkzeug bereits um die Version 5.0 von August 2005 handelt, scheint die Implementierung noch nicht ganz ausgereift zu sein. Auch nach intensivem Probieren ist es nicht gelungen, eine zuvor bearbeitete und abgespeicherte Datei mit Individuen zu bestücken. Ebensowenig kann man bereits bestehende Individuen oder andere Teile der Ontologie verändern. Lediglich eine neu erstellte Datei lässt sich bearbeiten, solange sie noch nicht abgespeichert ist.

Ferner scheinen einige Buttons zu fehlen, wie die Entfernen-Taste, um Individuen, die innerhalb der Session erstellt wurden, wieder zu löschen. Das Entfernen von Individuen funktioniert nur durch Markieren des Individuums in der Tripel-Ansicht und die Entfernen-Taste, was man von einem graphischen Werkzeug nicht erwarten würde. Auch das Abspeichern ist sehr kontra-intuitiv. In dem File-Menü heißt es *Save Ontology to local File*, wobei man erwarten würde, dass die komplette Ontologie mit den neu erstellten Individuen abgespeichert wird. Das ist aber nicht der Fall. Es werden nur die Klassen, mit den neuen Individuen abgespeichert. Man darf die Datei also keines Falls unter der zuvor geladenen Ontologie speichern, weil diese sonst überschrieben wird.

Da es nicht gelungen ist, eine abgespeicherte Datei weiter zu bearbeiten, scheint dieses Werkzeug gänzlich unbrauchbar zu sein. Erstaunlicherweise scheinen frühere Versionen, die aber heute nicht mehr erhältlich sind, besser funktioniert zu haben, wie Luxen (2003, S. 12ff, 15) zu entnehmen ist. Auch gab es wohl informative Tutorials, die im Internet aber ebenfalls nicht mehr zu finden sind. Eine Mailinglist zu dem Werkzeug für Rückfragen gibt es derzeit nicht, tröstlich ist allenfalls der Vermerk dazu auf der Home-Page: *coming soon* (Stand vom 20.08.2007).

6.4 OntoMat

Der OntoMat-Annotizer[39] ist eine Java-basierte Implementierung des umfangreichen Annotierungs-Frameworks CREAM, das unterschiedliche Annotierungsmethoden unterstützt. Mit OntoMat können bereits bestehende Webseiten manuell annotiert werden. Das Werkzeug beinhaltet einen Ontologie-Browser und einen HTML-Browser, der die zu annotierende Webseite darstellt (siehe Abbildung 6.4).

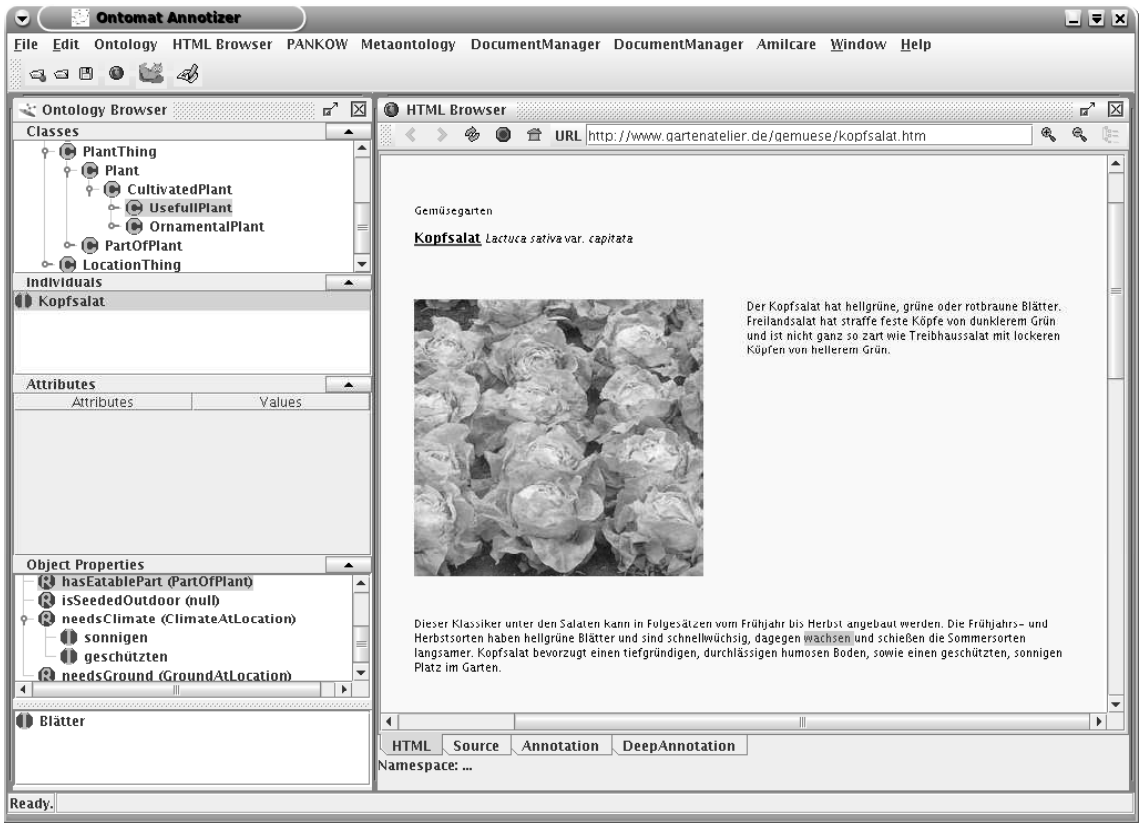


Abbildung 6.4: Die graphische Oberfläche des OntoMat

Wie auch bei SMORE können ausgewählte Teile der Webseite per Drag and Drop

[39] Vgl. OntoMat-Homepage: http://annotation.semanticweb.org/ontomat/index.html und Reif, 2006, S. 413

einer Ontologie-Klasse als Instanzen zugewiesen werden. Es können auch Klassen und Eigenschaften neu erstellt werden sowie zwei Ressourcen über eine Eigenschaft miteinander verknüpft werden. Der OntoMat ist sehr einfach und übersichtlich in der Handhabung. Ein kurzes aber prägnantes Tutorial gibt einen Überblick über die Möglichkeiten, die das Werkzeug bietet[40].
Mit importierten Ontologien scheint das Werkzeug aber nicht zurecht zu kommen. Bei der Gartenontologie, die sowohl die Time-Ontologie als auch die Mondontologie importiert, wird die Taxonomie im Ontologie-Browser nicht richtig dargestellt. Auch werden die meisten Eigenschaften, die die Klassen haben, nicht angezeigt. Der OntoMat funktioniert leider nur bei Ontologien, die keine weiteren Ontologien importieren.

6.5 Weitere Werkzeuge

Als ausgesprochen nützlich für das Debugging erwiesen sich die bereits erwähnten Validatoren und der Reasoner Pellet.
Der RDF-Validator des W3C[41] überprüft ein RDF-Dokument auf seine Validität und visualisiert es entweder als Tripel oder als Graph. Ist das Dokument nicht valide, gibt das Werkzeug detaillierte Fehlermeldungen mit Zeilenangaben an. Um das Dokument zu überprüfen kann man es entweder direkt in ein Eingabefenster kopieren oder den URL des Dokumentes angeben.
Ähnlich funktioniert der WonderWeb OWL Ontology Validator[42]. Die Eingabe erfolgt wie bei dem RDF-Validator. Zudem kann über Buttons angegeben werden, nach welchem OWL-Subtyp WonderWeb validieren soll: OWL-Light, OWL-DL oder OWL-Full. Der Validator gibt ausführlich Auskunft, wegen welcher Konstrukte es zu welchem OWL-Subtyp kommt und moniert sowohl Syntax- als auch Semantikfehler.
Pellet[43] ist ein open-source OWL-DL-Reasoner mit vollständiger Unterstützung für SHOIQ. Er ist also mit gleicher Ausdruckskraft wie OWL-DL ausgestattet. Er kann in Verbindung mit Jena, einem Java-Framework für Semantic Web-Anwendungen, oder über die schon erwähnte DIG-Schnittstelle benützt werden. Zudem gibt es eine Online-Demo, in die wie bei den Validatoren der Quellcode direkt eingegeben werden kann. Pellet kontrolliert die Konsistenz einer Ontologie, er klassifiziert die Taxonomie, überprüft mögliche Schlussfolgerungen und beantwortet ABox-Anfragen in RDQL oder SPARQL.

[40] OntoMat Tutorial: http://annotation.semanticweb.org/ontomat/tutorial.html
[41] Vgl. W3C Validation Service: http://www.w3.org/RDF/Validator/
[42] Vgl. WonderWeb OWL Ontology Validator: http://www.mygrid.org.uk/OWL/Validator
[43] Pellet-Homepage: http://www.mindswap.org/2003/pellet/

6.6 Resümee

Alle vier der beschriebenen Editoren hatten so große Mängel, dass die Gartenontologie bzw. die Annotation der Webseiten nicht ausschließlich mit einem Werkzeug erstellt werden konnte. Die Annotation erfolgte wegen der oben beschriebenen Probleme von Hand in einem Texteditor ohne jegliches Werkzeug. Die Ontologie wurde teils mit Protégé, teils mit SWOOP und gegen Ende der Entwicklung hauptsächlich im Texteditor erstellt. Bei der Bearbeitung im Texteditor waren die Ontologie-Editoren zur Visualisierung der Ontologie allerdings weiterhin in Gebrauch.

7 Der Vegi-Agent

Ein Software-Agent ist ein Programm, das zur Lösung eines Problems auf eigene Initiative autonom Aktionen ausführt. Kirn et al. (2006, S. 18) beschreiben einen Software-Agenten als: *an entity capable of action, and it takes its actions on behalf of another entity.* Ein Agent führt also Aktionen im Auftrag von jemand anders aus. Der Terminus *semantischer Agent* beschreibt die Vorstellung, dass es sich um ein Programm handelt, das inhaltsorientierte Auswertungen vornimmt.
Der Vegi-Agent hat die konkrete Aufgabe mit Hilfe der Ontologie aus den annotierten Seiten Fakten zu extrahieren. Für die Lösung dieser Aufgabe stehen ihm zwei Aktionsarten zur Verfügung: Er kann Anfragen an seine Wissensbasis stellen und er kann den Webservice aufrufen. Die Aktionsfolge ist dabei sehr starr: Der Agent stellt jede Anfrage an die erste Seite. Wenn er keine Antwort bekommt, geht er auf die nächste Seite, der er alle Anfragen stellt etc. Wenn er eine Antwort bekommen hat, braucht er die Anfrage der nächsten Seite nicht mehr zu stellen, statt dessen versucht er die noch fehlenden Inforamtionen zu finden. Hat er alle Informationslücken gefüllt, ist der Agent am Ziel und gibt die Informationen aus. Wenn er nicht alle Informationen gefunden hat, aber keine Seiten mehr zur Verfügung stehen, die er durchsuchen könnte, ist er ebenfalls am Ende seiner Aktionsfolgen.
Auch wenn es sich um eine recht simple Python-Implementierung handelt, verdient das Programm die Bezeichnung *Agent*, da es nach Kirn et al. (2006, 21ff) die charakteristischen Eigenschaften besitzt:

- Der Agent ist in eine Umgebung eingebettet.
  Alles, was außerhalb des Agenten liegt, wird als Umgebung bezeichnet. In dieser Studie sind das: die Eingabe, die fünf annotierten Webseiten und der Web Service. Diese Umgebung nimmt der Agent wahr, und er agiert in ihr. Die Umgebung des Vegi-Agenten ist eine statische Umgebung, weil sowohl die Formulierung als auch die Lösung des Problems erfolgen, ohne dass es Änderungen in ihr gibt.

- Der interne Prozess des Agenten bleibt dem Benutzer verborgen. Nur die externen Aktionen, die hier aus Eingabe und Ausgabe bestehen, kann man beobachten. Dadurch kann das Verhalten des Software-Agenten nur anhand der Qualität seiner Ergebnisse bewertet werden.

- Der Vegi-Agent ist autonom und arbeitet unabhängig von Benutzereingriffen; d.h. er hat bei der Ausführung seiner Aktionen ein gewisses Maß an Handlungsspielraum, der ihm erlaubt, seine eigenen Ziele zu verfolgen. So kann er beispielsweise entscheiden, den Mondkalender nicht aufzurufen, wenn er kein Saatzeit-Intervall gefunden hat. Oder er wählt nur die erfolgversprechenden Seiten für seine Anfragen aus. Sucht er nach Informationen über Tomaten, befragt er nicht eine Seite, auf der es um Kopfsalat geht.

- Mit Hilfe der Ontologie kann der Agent Schlussfolgerungen ziehen. Diese Fähigkeit braucht er z.B., um aus den Informationen des Mondkalenders die für die Pflanze günstigen Saattermine zu filtern.

Der Agent ist ferner proaktiv, was bedeutet, dass er Aktionen aufgrund eigener Initiative auslöst. Diese Eigenschaft kommt bei dem Aufruf des Web Services zur Geltung. Im übertragenen Sinne wird der Software Agent mit diesem Aufruf aus eigener Initiative zum Kunden des Web Services. Proaktivität lässt ihn nach Kirn et al. (2006, 24) bereits in die Nähe eines *intelligenten* Agenten rücken. Die anderen Eigenschaften eines intelligenten Agenten wie Lernfähigkeit, soziale Eigenschaften, die ihn befähigen würden mit anderen Agenten zu kommunizieren und Reaktivität sind ihm allerdings nicht eigen, wobei letztere Eigenschaft bei dem Vegi-Agenten auch nicht angestrebt ist. Ein reaktiver Agent nimmt jede Änderung innerhalb der Umgebung wahr und reagiert darauf unmittelbar – eine Eigenschaft, die für Terminplaner z.B. sehr nützlich ist. Würde aber in der Umgebung des Vegi-Agenten die Information auftauchen, dass beispielsweise Kopfsalat doch im Schatten gut gedeiht, so würde es ausreichen, wenn der Agent diese Information bei der nächsten Anfrage, die diese Änderung betrifft, berücksichtigt. Es macht aber keinen Unterschied, ob er sie auch schon vorher gekannt hat oder nicht.

7.1 Reasoning des Agenten

Der Vegi-Agent ist ein wissensbasierter Agent, d.h., dass er sowohl mit einer Wissensbasis ausgestattet sein muss, aus der er Informationen beziehen kann, als auch mit einem Reasoner, der ihn befähigt, zusätzliche Informationen selbst zu erschließen. Die Ontologien liefern ihm z.B. die Prämissen:

- Jedes Blattgemüse soll bei zunehmendem Mond gesät werden.

- An <Datum> ist zunehmender Mond.

Daraus folgt die Konklusion:

- An <Datum> soll Kopfsalat gesät werden.

Das logische Schließen wird durch die logischen Relationen ermöglicht, aus denen die Ontologie aufgebaut ist. Aufgrund dieser Wissensrepräsentation kann der Reasoner durch Inferenz neues Wissen ableiten.
Es gibt bereits diverse Inferenzmaschinen wie z.B. Pellet[44], die auf Beschreibungslogik basieren, und damit für die hier benötigten Zwecke in Frage kommen. Im Rahmen der Studie wurde allerdings kein externer Reasoner eingebunden, sondern vorläufig eine eigene Funktion implementiert, die die Schlussfolgerung zieht.

7.2 Vorüberlegungen zur Implementierung

7.2.1 Finden der Seiten

Im Semantic Web würde ein Agent das Netz über eine ontologie-erweiterte Suchmaschine durchforsten wie es sie heute z.B. schon mit Swoogle[45] gibt. Da hier in einer künstlichen Umgebung gearbeitet wurde, stehen die Links zu den annotierten Seiten in einer Konfigurationsdatei. Die Schritte des Retrievals im Web sind nicht berücksichtigt, sind aber auch nicht Gegenstand dieser Studie. Der Agent braucht die Adresse der RDF-Datei der Gemüsegarten-Seite, auf der die Links zu den annotierten Seiten der einzelnen Gemüsesorten stehen. Diese Links verfolgt er selbst, weshalb eine Angabe der Kopfsalat-, Tomate- und Buschbohnenseite nicht nötig ist. Ebenso muss er die Adresse zu der Seite mit den Mischkulturen haben. Für den Webservice bekommt er die URI der OWL-S- Beschreibung. Dort findet er alle benötigten Informationen, um diesen Service zu nutzen.

7.2.2 Verbindung von Annotation, Ontologie und Agent

Der Agent bezieht sein Wissen aus der Ontologie, die ihm gegeben ist. Die Ontologie ist aber, wie in Kapitel 4 und 5 bereits beschrieben, nicht mit Instanzen bestückt. Die einzelnen Instanzen wie *Kopfsalat, Tomate* etc. werden erst mit der Annotation der Webseiten ins Leben gerufen. Dadurch weiß der Agent erst, wenn er die annotierten Webseiten sieht, dass beispielsweise ein Kopfsalat eine Pflanze ist. Das hat zur Folge, dass der Agent einen String als Eingabe erhält, von dem er zunächst nicht weiß, ob es sich um ein Gemüse handelt oder nicht. Ist der String aber in einer der annotierten Seiten zu finden, so weiß er über die Ontologie, zu welcher Klasse dieser String gehört und kann, wenn es ein Gemüse ist, die gewünschten Informationen extrahieren. Ansonsten ist die Ausgabetabelle leer.

[44] Vgl. Parisa/Sirin 2004

[45] Swoogle ist eine Suchmaschine für Semantic Web-Dokumente inklusive Ontologien. Swoogle-Homepage: http://swoogle.umbc.edu/

7.2.3 SPARQL

Der Agent braucht eine Abfragesprache, mit der er auf die Informationen in der Ontologie zugreifen kann. Hierfür wird die Sprache SPARQL[46] verwendet, die im Juni 2007 als W3C-Empfehlung verabschiedet wurde. SPARQL ist speziell dazu entwickelt, Anfragen an RDF-Daten auszudrücken. Die Abfragesprache basiert darauf, die RDF-Tripel zu matchen, wobei Subjekt, Prädikat und/oder Objekt durch eine Variable ersetzt werden können. Findet SPARQL einen exakten Match zu der Anfrage, können die Werte der Variablen ausgegeben werden. Braucht man z.B. aus einem RDF-Tripel wie in Listing 7.1 die Information, zu welcher Gartenontologie-Klasse die Ressource *Tomate* gehört,

```
1     <rdf:Description rdf:about="Tomate">
2         <rdf:type rdf:resource="&garden;Tomato"/>
```

Listing 7.1: RDF-Tripel

so würde dazu die Abfrage folgendermaßen aussehen:

```
1 SELECT  ?class
2 WHERE {
3     <http://www.it-devel.de/semanticWeb/res/annotation/Tomate>
4     rdf:type
5     ?class
6     }
```

Listing 7.2: Beispiel einer SPARQL-Abfrage

SPARQL gibt dazu folgende Ausgabe: `&garden;#Tomato`, was die Adresse der Ontologie und die Klasse ausdrückt. Diese Anfrage liefert nur einen Treffer zurück. Wenn die Ressource aber zu mehreren Klassen gehört, gibt das Programm sämtliche Treffer aus. Hinter `SELECT` (Zeile 1) stehen alle Variablen, die SPARQL als Ergebnis ausgibt. `WHERE` (Zeile 2) enthält das Muster des RDF-Graphen, das gegen die RDF-Daten gematcht wird.
Gibt es mehrere Ergebnisse, können diese gefiltert werden. Ein SPARQL-Filter beschränkt die Ergebnismenge auf jene Ergebnisse, bei denen der Filter-Ausdruck wahr ist. Die unten stehende Anfrage filtert z.B. aus allen Ressourcen, die nach der Gartenontologie eine Pflanze sind, die Tomaten heraus:

```
1 SELECT ?obj
2 WHERE {
3     garden:Plant ?pred ?obj.
4     FILTER regex(?obj, "#Tomate$")}
```

Listing 7.3: Beispiel eines SPARQL-Filters

[46]SPARQL Query Language: http://www.w3.org/TR/rdf-sparql-query/

Mit SPARQL lassen sich neben einfachen Graphen auch ganze Gruppen von Tripeln darstellen. Möchte man z.B. den Anfang eines Saatzeitraums von einer bestimmten Gemüsesorte erfragen (siehe RDF-Daten in Listing 7.4), muss man mehrere RDF-Beschreibungen miteinander kombinieren.

```
1     <rdf:Description rdf:about="Tomate">
2         <garden:isSeeded rdf:resource="SaatzeitSalatDrinnen"/>
3     </rdf:Description>
4
5     <rdf:Description rdf:about="SaatzeitSalatDrinnen">
6         <time:hasBeginning rdf:resource="EndeFebruar"/>
7     </rdf:Description>
```

Listing 7.4: Anfang eines Saatzeitraums

Die einzelnen Tripel in der SPARQL-Abfrage werden von unten nach oben gelesen und mit einem Punkt verbunden:

```
1 SELECT ?seedingBegin
2 WHERE {
3     ?seedingTime time:hasBeginning ?seedingBegin .
4     <http://www.it-devel.de/semanticWeb/res/annotation/Tomate>
5         garden:isSeeded ?seedingTime
6     }
```

Listing 7.5: SPARQL-Anfrage für den Anfang eines Saatzeitraums

In den Zeilen 4 und 5 steht der Graph, der in den RDF-Daten (Listing 7.4) den ersten drei Zeilen entspricht. Die SPARQL-Variable `?seedingTime` wird an Zeile 3 übergeben, die das zweite RDF-Tripel (Zeilen 5-7 der RDF-Daten) matcht und als Ergebnis den Wert der Variable `seedingBegin` zurückgibt – hier `EndeFebruar`.
Da SPARQL eigentlich für Anfragen an RDF-Dateien konzipiert ist und weniger für Ontologieanfragen entwickelt wurde, wird keine Vererbung unterstützt. Dadurch sind Anfragen mit vererbten Eigenschaften sehr lang und umständlich zu implementieren. Man muss mit der `rdfs:subClassOf`-Eigenschaft den Ontologiebaum soweit nach oben verfolgen, bis der Knoten, für den die gewünschte Eigenschaft definiert wurde, erreicht ist.

7.3 Implementierung des Agenten

Der Agent besteht aus unterschiedlichen Modulen. `Sparql.py` ist dabei eine zentrale Schnittstelle zwischen den Modulen des Agenten und den Ontologien, den RDF-Dokumenten bzw. der OWL-S-Beschreibung des Web Services. Mit den Modulen `vegi.py, webService.py` und `moon.py` bezieht der Agent Informationen über das

Gemüse, über den Web Service und über die Monddaten eines Tages aus den zur Verfügung stehenden Ressourcen (siehe Abbildung 7.1).

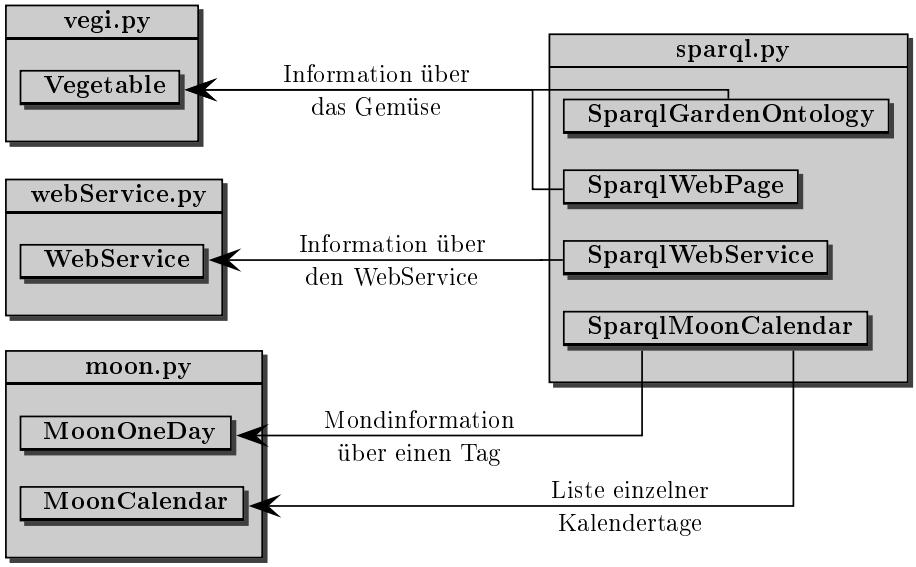


Abbildung 7.1: Zugriffe auf das SPARQL-Modul

Abbildung 7.2 zeigt das Zusammenspiel der einzelnen Komponenten des Agenten, auf die in den folgenden Kapiteln näher eingegangen wird.

7.3.1 Die Gemüse-Klasse

Die Klasse `class Vegetable` besteht fast nur aus `set-` und `get`-Funktionen, die die unterschiedlichen Informationen über die Gemüsesorte beziehen bzw. ausgeben. Diese Informationen erhält die Klasse von der SPARQL-Klasse. Neben den Informationen, die unmittelbar ausgegeben werden wie Bodenbeschaffenheit, Standort, Saatzeit etc. werden noch einige interne Abfragen benötigt. Die Saatzeit wird beispielsweise mit der Angabe *Ende Februar bis Anfang April* ausgegeben. Mit dieser Angabe kann der Agent nicht überprüfen, ob ein Datum aus dem Mondkalender in dieses Intervall fällt. Deshalb muss der Agent – um bei dem Beispiel zu bleiben – zusätzlich folgende Abfragen stellen:

- Was für ein `xsd:gMonth` ist *Februar* bzw. *April*? (Funktionen `setMonthDateTime`, `getGMonthOfMonth`)
- Am wievielten des Monats beginnt das Intervall *Ende* des Monats, hier *Ende Februar*, und am wievielten endet das Intervall *Anfang* des Monats, hier *Anfang April*? Diese Angaben liefert die Ontologie im `xsd:gDay`-Format. (Funktionen `setPartOfMonth, getGDayFromBeginOrEnd`)

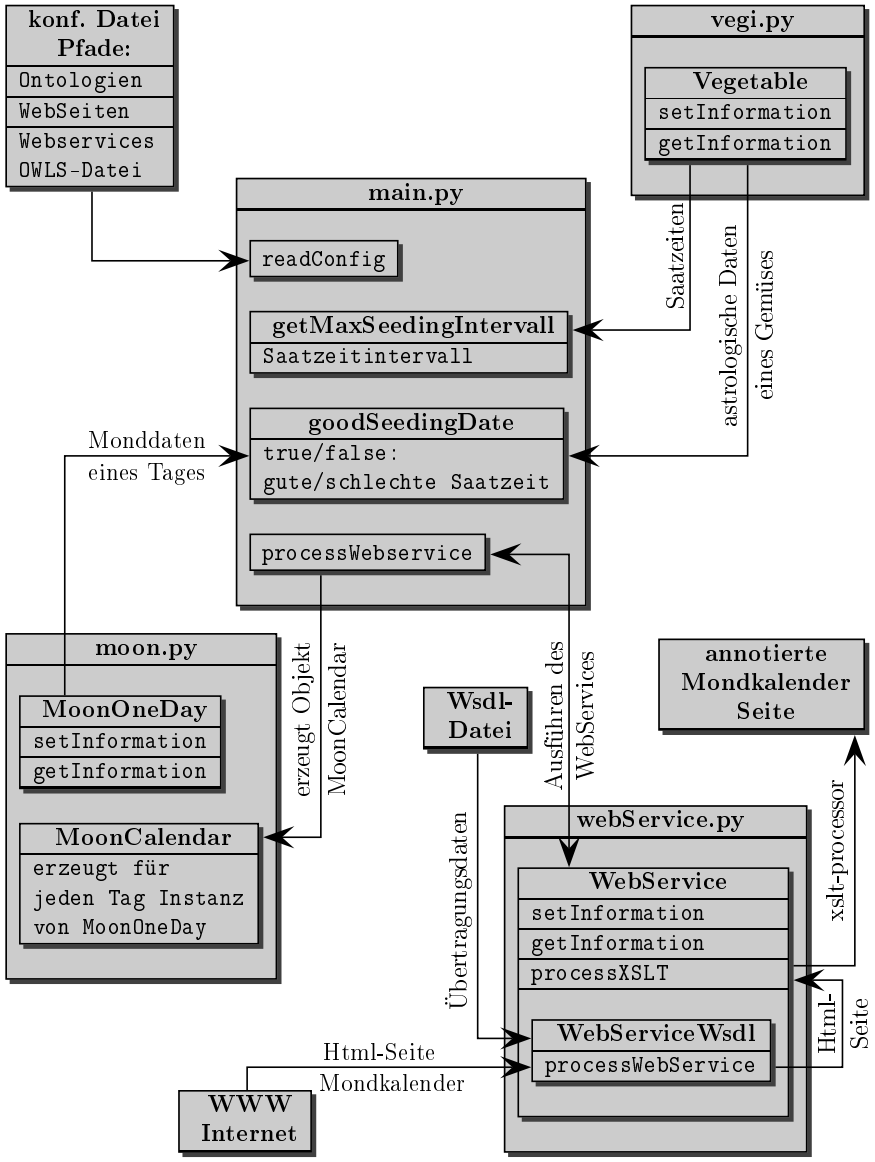


Abbildung 7.2: Zusammenspiel der Komponenten des Agenten

- Um welches Gemüse handelt es sich, sprich: zu welcher Klasse der Ontologie gehört es? (Funktion `setType`)

- Um die Tage angeben zu können, die nach dem Mondkaldender besonders günstig für die Aussaat einer bestimmten Gemüsesorte sind, muss der Agent wissen, welche Mondsternzeichen und welche Mondphase das Gemüse braucht. (Funktionen `setBestSeedingTimeSignOfZodiac`, `setBestSeedingTimeMoonPhase`)

7.3.2 Die Mond-Klassen

Für die Angaben zu dem Mondkalender wurden zwei Klassen implementiert. Die Klasse `class MoonCalendar` bekommt den annotierten Mondkalender eines Monats

und zerlegt ihn in einzelne Tage. Für die einzelnen Tage wird jeweils ein `MoonOneDay`-Objekte erzeugt. Die `MoonOneDay`-Klasse ist ähnlich aufgebaut wie die Klasse `class Vegetable`. Auch sie besteht hauptsächlich aus `set-` und `get`-Funktionen, die aus der SPARQL-Klasse Informationen beziehen und zurückgeben. Relevante Angaben in dem Mondkalender sind: das Sternzeichen, in dem der Mond sich an dem Tag befindet und das Alter des Mondes innerhalb der Mondphase. Bei letzterer Information handelt es sich um eine Angabe im `xsd:decimal`-Format. Ist die Zahl größer als 14 ist der Mond zunehmend, ist die Zahl kleiner oder gleich 14 ist der Mond nicht zunehmend. Die Variable `self.waxingMoon` hat einen boolschen Wert. Damit das Datum von Maschinen interpretiert werden kann, gibt es in der Klasse die Funktion `setMcXsdDate`, die sich das Datum im `xsd:Date`-Format holt.

7.3.3 Die Web Service-Klassen

Um den Web Service zu nutzen braucht der Agent Informationen aus der OWL-S-Ontologie und aus dem WSDL-Dokument. Dafür sind die Klassen `class WebService` und `class WebServiceWsdl` zuständig. Das Prinzip der `WebService`-Klasse entspricht den oben beschriebenen Klassen mit `get-` und `set`-Funktionen. Sie bekommt den URI der OWL-S-Datei, die den Web Service beschreibt und enthält folgende Funktionen, die im Großen und Ganzen alle dazu dienen, Informationen aus der SPARQL-Klasse zu holen:

- `setAtomicProc`: holt den Namen des Atomic Process.
- `setWsdlDoc`: holt die URI des WSDL-Dokuments.
- `setWsdlService`: holt den zu dem Atomic Process korrespondierenden WSDL-Service.
- `setPortType`: holt den zu dem Service gehörenden Port, der zum einen die http-Adresse enthält und zum anderen das Binding von dem Service zu der Web Service-Operation.
- `setOperation`: holt die Operation, in der das Eingabeformat und das Ausgabeformat spezifiziert wird.
- `setInput`: verbindet das Eingabe-Nachrichtenformat des Web Services einerseits mit der OWLS-Process-Beschreibung der OWL-S-Ontologie und ferner mit dem WSDL-Eingabeformat. Holt die OWL-S-Parameter `month` und `year` aus der Process-OWLS-Ontologie.
- `setOutput`: die Funktion funktioniert in Analogie zu setInput. Sie holt außerdem den XSLT-Transformationsstring aus dem OWL-S-Dokument.

- `processXSLT`: bei vorhandenem XSLT-Transformationsstring wird ein HTML-Dokument als Parameter übergeben und mit Hilfe des xsltproc-Programms verarbeitet. Xsltproc bekommt den zu transformierenden String, das HTML-Dokument, über seine Standardeingabe und gibt den XSL-Ergebnisbaum, das RDF-Dokument, über seine Standardausgabe zurück. Das neu generierte RDF-Dokument wird in einer temporären Datei abgespeichert.

Die Klasse `WebServiceWsdl` dient der tatsächlichen Ausführung des Web Service. Sie bekommt den URI der WSDL-Beschreibung, für deren Auswertung die Python-Module des ZSI-Toolkit[47] verwendet werden. Neben einigen Hilfsfunktionen, die beispielsweise kontrollieren, ob alle benötigten Angaben aus der WSDL-Beschreibung hervorgehen, enthält die Klasse die Funktion `processWebService`, die die Ausführung des Web Service durchführt.

7.3.4 Die SPARQL-Klassen

Die Klasse `class Sparql` importiert u.a. das Modul `rdflib`. Die RDFLib ist eine Python-Bibliothek, die eigens für die Verarbeitung von RDF-Dokumenten entwickelt wurde[48]. Neben vielen anderen nützlichen Features unterstützt sie SPARQL, wodurch komfortabel innerhalb einer Pythonimplementierung mit der RDF-Abfragesprache gearbeitet werden kann. In die neueste Version (2.4.0) der RDFLib wurde eine neue SPARQL-Klasse eingeführt, die mit der übrigen Implementierung noch nicht richtig zusammen zu arbeiten scheint. Das betrifft auch die query-Funktion der Graphen, die auf diese Klasse umgestellt wurden. Deswegen wurde auf die ältere Version 2.3.3 zurückgegriffen, in der es diese Klasse noch nicht gibt.
Die SPARQL-Klassen sind unmittelbar auf die hier benützten Ontologien bezogen. Wollte man Informationen aus Webseiten extrahieren, die mit einer anderen Gartenontologie annotiert wurden, müsste beispielsweise die Klasse `SparqlWebPages` neu geschrieben werden. Alles andere könnte beibehalten werden. Die Kompatibilität zu anderen Web Services ist aber gegeben, weil hier die Annotation immer auf OWL-S, also auf der gleichen Ontologie, beruht.
Die Klasse `Sparql` ist eine Basisklasse zu diversen Unterklassen, die Anfragen an unterschiedliche Ressourcen stellen. Die Basisklasse selbst bekommt die Pfade der benötigten Ontologien und enthält zwei Funktionen, die mit Hilfe der RDFLib entweder ein Ergebnis (`querySelect1`) oder mehrere Ergebnisse (`querySelect`) einer SPARQL-Anfrage zurückgeben.

[47] Vgl. ZSI Developer's Guide: http://pywebsvcs.sourceforge.net/zsi.html
[48] Vgl. RDFLib-Homepage: http://rdflib.net/

Die Klasse `class SparqlWebPages`

Die Klasse erhält das vom Benutzer ausgewählte Gemüse und die Adressen der annotierten Seiten. Die Funktion `searchForVegi` sucht auf den angegebenen annotierten Webseiten nach dem Gemüse und speichert den Link in einer Liste, falls das Gemüse vorkommt. Für diese Suche wird bereits eine SPARQL-Anfrage an das RDF-Dokument gestellt. An jede ausgesuchte RDF-Seite werden anschließend alle benötigten Anfragen gestellt; das sind für den Benutzer Anfragen zu:

- der Bodenbeschaffenheit (`groundQuery`)
- zu Standortangaben (`climateQuery`)
- zu Mischkulturen (`goodNeighbourQuery, badNeighbourQuery`)
- zu der Saatzeit (`seedingTimeQuery`)

Wie in 7.3.1 bereits ausgeführt wurde, sind für die interne Weiterverarbeitung weitere Anfragen nötig:

- `typeQuery` bekommt ein Gemüse und erfragt die Ontologie-Klasse, zu der es gehört.
- `monthDateTimeQuery` bekommt eine Saatzeit wie *Ende Februar* und erfragt die Klassenzugehörigkeit des Monats.
- `def gMonthQuery` bekommt einen Monat und bestimmt den `xsd:gMonth` zu dem Monat.
- `partOfMonthQuery` bekommt eine Saatzeit wie *Ende Februar* und ermittelt die dazugehörige PartOfMonth-Klasse der Ontologie.
- `gDayQuery` bekommt einen Monatsteil und die Angabe, ob es sich um den Anfangs- oder Endzeitpunkt des Intervalls handelt. Die Funktion gibt im `xsd:gDay`-Format den ersten bzw. letzten Tag einer Saatzeit zurück.

Die Klasse `class SparqlWebService`

Die Klasse stellt Anfragen an die OWL-S-Beschreibung des Web Service und bekommt dafür die Pfadangabe der Beschreibung. Die Anfragen sind hier sehr einfach. Erfragt wird der Atomic Process (`atomicProcessQuery`), das korrespondierende WSDL-Dokument (`wsdlDocQuery`), die Adresse des Web Services (`wsdlServiceQuery`), der Port-Type (`portTypeQuery`), die Operation(`operationQuery`) sowie Input und Output (`inputQuery, outputQuery`).

Die Klasse `class SparqlMoonCalendar`

Die Klasse bekommt die Adresse der RDF-Datei über den Mondkalender – also das Ergebnis des XSL-Stylesheets. In der Funktion `signOfZodiacQuery` wird ein Mondkalender-Datum als String übergeben und das Mondsternzeichen des Kalendertages abgefragt oder genauer gesagt: einer Mondontologie-Klasse zugewiesen. Die Funktion `ageOfMoonQuery` bekommt ebenfalls ein Datum, zu dem sie das Mondphasen-Alter des Mondes im `xsd:decimal`-Format zurück gibt. Mit der Funktion `mcDateQuery`, die ebenfalls ein Datum als String erhält, wird eine Anfrage gestellt, die das Datum im `xsd:date`-Format zurückgibt. Die Funktion `mcOneDayQuery` sucht jede `rdf:Description`, die ein neues Datum liefert, heraus. Damit wird ein Mondkalendermonat in einzelne Tage zerlegt.

Die Klasse `class SparqlGardenOntology`

Die Klasse stellt Anfragen an die Garten-Ontologie. Mit ihren Funktionen wird in Erfahrung gebracht, bei welcher Mondphase und bei welchem Mondsternzeichen eine Pflanze gesät werden soll. Hier stellt sich das Problem, dass SPARQL keine Vererbung unterstützt. Der Hierarchiebaum von Tomate sieht z.B. folgendermaßen aus: {`Tomato, FruitVegetable, Vegetable, UsefulPlant, ...`}. Die Information, bei welchem Mond- und Sternenstand die Pflanze am besten gesät wird, ist in der Klasse `FruitVegetable` gegeben und wird an die Tomate vererbt. Teilweise gibt es aber zwischen der untersten Gemüseklasse und der gesuchten Klasse der Gemüsegruppe noch einige Zwischenklassen. Der Agent muss also zunächst die Oberklasse des Gemüses finden, die entweder `FruitVegetable, RootVegetable, LeafVegetable` oder `BlossomVegetable` heißt. Da diese Klassen alle eine direkte Unterklasse von der Klasse `Vegetable` sind, sucht der Agent so lange in einem rekursiven Aufruf nach der Oberklasse, bis die nächste Oberklasse die Klasse `Vegetable` ist. Damit befindet er sich in der gesuchten Klasse. Diese Anfrage stellt die Funktion `vegiSuperClassQuery`.
Umständlich in der Implementierung ist auch die Funktion `bestSeedingTimeSignOfZodiacQuery`, die die Oberklasse der Gemüsegruppe bekommt – z.B. `FruitVegetable` und eine Liste von günstigen Sternzeichen zurückgibt – z.B. `Aries, Leo, Sagittarius`. Die Ontologie ist so implementiert, dass eine Schnittmenge zwischen der günstigen Mondphase und der Vereinigungsmenge der drei günstigen Sternzeichen gebildet wird. Schnittmengen und Vereinigungsmengen werden von SPARQL als verkettete Listen gelesen. Wenn man also drei Elemente in einer Vereinigungsmenge hat, so muss man drei Mal Tripel mit den Prädikaten `rdf:first` und `rdf:rest` aufstellen. Die Anfragen werden dadurch sehr lang und für Menschen schwer lesbar. Es ist nicht möglich eine rekursive Funktion zu implementieren, die

die verkettete Liste durchläuft, weil die zurückgegebenen anonymen Klassen bei jedem Aufruf eine neue ID haben und deshalb bei einer erneuten Anfrage nicht wiederverwendet werden können. Die folgenden Listings zeigen den Ontologie-Code der Klasse `FruitVegetable` und die entsprechende SPARQL-Anfrage, die die drei günstigen Sternzeichen extrahiert.

```
<owl:Class rdf:about="&garden;FruitVegetable">
  <rdfs:subClassOf>
    <owl:Class rdf:about="&garden;Vegetable">
    </owl:Class>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&garden;bestSeedingTime"/>
      <owl:someValuesFrom>
          <owl:Class>
              <owl:intersectionOf rdf:parseType="Collection">
                <owl:Class rdf:about="&moon;WaxingMoon"/>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <owl:Class rdf:about=
                            "&moon;MoonInAries"/>
                        <owl:Class rdf:about=
                            "&moon;MoonInLeo"/>
                        <owl:Class rdf:about=
                            "&moon;MoonInSagittarius"/>
                    </owl:unionOf>
                </owl:Class>
              </owl:intersectionOf>
          </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Listing 7.6: OWL-Code der Klasse `FruitVegetable`

```
SELECT  ?sign1 ?sign2 ?sign3
            WHERE {
            ?anonymCl4 rdf:first ?sign3 .
            ?anonymCl3 rdf:rest ?anonymCl4   .
            ?anonymCl3 rdf:first ?sign2 .
            ?parseType2 rdf:rest ?anonymCl3 .
```

```
7               ?parseType2 rdf:first ?sign1     .
8               ?anonymCl2 owl:unionOf ?parseType2 .
9               ?anonymCl1 rdf:first ?anonymCl2 .
10              ?parseType1 rdf:rest ?anonymCl1 .
11              ?class owl:intersectionOf ?parseType1    .
12              ?anonymRestr owl:someValuesFrom ?class   .
13              garden:FruitVegetable rdfs:subClassOf ?anonymRestr
14              }
```

Listing 7.7: SPARQL-Abfrage für die drei Sternzeichen

7.3.5 Das Main-Programm

In der Datei `main.py` befinden sich alle Funktionen, die die oben beschriebenen Komponenten zusammenfügen.

- `readConfig` liest die Konfigurations-Datei ein, in der alle benötigten Pfade angegeben sind: die Pfade der Ontologien, die Pfade der Webseiten und den Pfad der OWL-S-Beschreibung für den Web Service.

- `getMaxSeedingInterval` bekommt ein Objekt der Klasse Vegetable und ein Jahr. Die Funktion sucht das größtmögliche Saatzeitintervall in Bezug auf die `SeedingtimeIndoor` und die `SeedingtimeOutdoor`. Sie gibt den frühesten und den spätesten Saattermin für dieses Jahr zurück.

- `goodSeedingDate` bekommt ein Datum, ein Gemüse sowie den Anfangs- und Endzeitpunkt eines Saatintervalls. Die Funktion entscheidet, ob das Datum innerhalb des Saatintervalls liegt und nach dem Mondkalender günstig für die Aussaat ist. Die logische Schlussfolgerung, die der Agent ziehen muss, ist in dieser Funktion implementiert.

- `processWebService` bekommt den Web Service sowie einen Anfangs- und einen Endzeitpunkt für die Saatzeit. Die Funktion führt den Web Service für alle benötigten Monate aus – also für die ganze Saatzeit, und gibt ein MoonCalendar-Objekt für das gesamte Saatintervall zurück.

- Ein- und Ausgabefunktionen: Der Agent ist bisher noch ein Kommandozeilen-Programm, in das der Benutzer ein Gemüse und die Jahreszahl, in der gesät werden soll, eingeben kann. Die Ausgabe erfolgt tabellarisch:

```
1 Bitte ein Gemuese eingeben:  Kopfsalat
2 Bitte Aussaatjahr eingeben [Default 2007]:
3 Bitte etwas Geduld...
```

```
4
5  Gemuese:                    | Kopfsalat
6  ================================================================
7  Boden:                      | HumoserBoden, TiefgruendigerBoden,
8                              | DurchlaessigerBoden
9  ----------------------------------------------------------------
10 Standort:                   | WindgeschuetztesKlima, SonnigesKlima
11 ----------------------------------------------------------------
12 Gute Nachbarn:              | Rettich, Bohne, Chicoree,
13                             | Schwarzwurzel, Erbse, Moehren,
14                             | Kohlrabi, Lauch, Tomate,Gurke,
15                             | Fenchel, Radieschen, Zwiebel, Kohl
16 ----------------------------------------------------------------
17 Schlechte Nachbarn:         | Petersilie, Kresse, Sellerie
18 ----------------------------------------------------------------
19 Saatzeit drinnen:           | EndeFebruar bis EndeMaerz
20 ----------------------------------------------------------------
21 Saatzeit draussen:          | April bis EndeJuli
22 ----------------------------------------------------------------
23 Guenstige Tage zum saeen:   | 18-02-2007, 26-02-2007, 27-02-2007,
24                             | 27-3-2007, 26-3-2007, 23-04-2007,
25                             | 22-04-2007, 30-05-2007, 19-05-2007,
26                             | 20-05-2007, 26-06-2007, 17-06-2007,
27                             | 16-06-2007, 27-06-2007, 24-07-2007
28 ----------------------------------------------------------------
```

Listing 7.8: Ein- und Ausgabe des Vegi-Agenten

Für besseren Benutzerkomfort wäre eine Ein- und Ausgabe über eine graphische Oberfläche denkbar, bei der der Anwender die Informationen anklicken kann, die er braucht.

7.3.6 Agenten im Semantic Web

Durch den modularen Aufbau des Semantic Web bleibt ein Semantic Web-System relativ flexibel. Bei der vorliegenden Studie können beispielsweise ohne weitere Änderungen sämtliche Gemüsesorten neben Kopfsalat, Buschbohne und Tomate aufgenommen werden. Dafür müssten lediglich entsprechende Webseiten annotiert werden, was in der Masse zwar einen Aufwand bedeutet, aber dennoch sehr viel weniger zeitintensiv ist als das Erstellen der HTML-Webseite, bei der man Formulierung, Format etc. mitberücksichtigen muss. Wollte man eine andere Gartenontologie bei der Annotation benützen, so müssten die entsprechenden SPARQL-Anfragen, die

an die neue Ontologie gestellt würden, im Agenten neu implementiert werden. Alles andere könnte beibehalten werden. Auch der Web Service ließe sich relativ leicht austauschen. Wenn es bereits eine OWL-S- und WSDL-Beschreibung gäbe, müsste nur das XSL-Stylesheet neu geschrieben werden.
Sehr aufwändig sind dagegen jegliche Änderungen innerhalb der Ontologie. Möchte man neue Informationen z.B. über Schädlinge oder die Gemüseernte mitaufnehmen, müsste die Ontologie erweitert werden, was schon allein viel Zeit kostet. Zudem muss bei jeder Erweiterung penibel darauf geachtet werden, dass keine bestehenden Klassen und Eigenschaften verändert werden, weil sonst schon bestehende Annotationen falsch würden und die Anfragen des Agenten nicht mehr auf die Ontologie passen würden. Der Aufwand von Veränderungen in der Ontologie ist ausgesprochen groß, weil sowohl der Agent als auch die Annotation unmittelbar darauf zugeschnitten sind.
Je nach eingegebenem Gemüse variiert die Laufzeit des Agenten. Das hängt vor allem damit zusammen, dass die Gemüsesorten unterschiedlich lange Saatzeitintervalle haben. Dabei gilt: je länger das Saatzeitintervall umso länger die Laufzeit. Kopfsalat kann beispielsweise von Februar bis Juli gesät werden, was zur Folge hat, dass der Agent für alle sechs Saatzeitmonate den Mondkalender aufrufen muss, um die günstigen Saattermine zu finden. Auch wenn noch einige Verbesserungen bei der Implementierung möglich und nötig wären, braucht der Agent für diese sechs Web-Anfragen mit allen übrigen nötigen Ermittlungen eine knappe Minute. Verglichen mit der Zeit, die ein menschlicher Benutzer dafür brauchen würde, ist das durchaus eine akzeptable Leistung.

8 Fazit und Ausblick

In dieser Arbeit wurde exemplarisch ein System vorgestellt, das mit Hilfe von Semantic Web-Technologien Wissen aus dem WWW erschließt. Im Rahmen einer Studie entstand ein Modell des Semantic Web, in dem ein Agent operieren kann.
Die Studie behandelt das Wissensgebiet der Hobby-Gärtnerei, das, wie sich gezeigt hat, in Bezug auf Semantic Web-Technologien noch gänzlich unerschlossen ist. Es waren weder annotierte Seiten noch Ontologien zu finden. Um ein möglichst realistisches Szenario zu simulieren, wurde das Wissensgebiet so gewählt, dass das gesuchte Wissen aus unterschiedlichen, von einander unabhängigen Quellen wie Garten-Hypertexten, einem Mondkalender etc. stammt und in möglichst unterschiedlicher Form vorliegt, z.B. in freiem Text oder in Tabellen.
Einige der benötigten Ontologien wie die Gartenontologie und die Ontologie für den Mondkalender wurden von Grund auf neu erstellt, während die Beschreibung der Zeitverhältnisse aus der Time-Ontologie des W3C mit einigen Erweiterungen übernommen werden konnte. Dabei wurde deutlich, dass es sowohl schwierig ist Ontologien zu entwickeln als auch sie anzuwenden. Je komplexer die Domäne ist und je mehr Freiraum sie für Interpretation und freies Assoziieren bietet, desto schwieriger ist es, das Gebiet in starren Termen zu beschreiben. Das Erstellen und Bearbeiten einer gut funktionierenden Ontologie erfordert sehr viel Fachwissen und Erfahrung.
Die Annotation der Webseiten erfolgte teils manuell von Hand und teils automatisch mit Hilfe eines XSL-Stylesheets. Es hat sich gezeigt: je formaler die Struktur des Textes ist, um so einfacher gestaltet sich die Annotation und umso weniger bleibt es der persönlichen Ansicht des Annotators überlassen, was relevante Information ist. Bei mehreren Seiten mit ähnlich strukturiertem freiem Text wie bei den Gemüseseiten ist die erste Annotation aufwändig. Für alle weiteren Seiten müssen dann nur noch die Werte manuell ausgetauscht werden. Informationen, die in Tabellen vorliegen, lassen sich gut in Containern oder Collections darstellen. Tabellen, die sich mit unterschiedlichen Werten wiederholen, können, wie bei der Annotation des Mondkalenders, mit einem Stylesheet automatisch annotiert werden.
Der Agent nutzt die Ontologien und die RDF-Dokumente, um an ihn gestellte Anfragen zu beantworten. Obwohl der Agent aus einer recht einfachen Python-Implementierung besteht, könnte er bereits nutzbringend eingesetzt werden, wenn es mehr annotierte Gemüseseiten gäbe. Der Zugriff auf die Ressourcen erfolgt über die Abfragesprache SPARQL, die primär für RDF entwickelt worden ist und daher lei-

der keine Vererbung unterstützt. Das macht die Abfrage von Ontologien gegebenfalls recht umständlich. Für die Implementierung existieren bereits einige einigermaßen ausgereifte Programmierwerkzeuge wie RDFLib und ZSI, mit denen man komfortabel eine Semantic Web-Umgebung nützen kann.
Die Studie hat gezeigt, dass die Semantic Web-Technologien die Möglichkeiten des Information Retrieval deutlich erweitern:

- Der Vegi-Agent ist in der Lage festzustellen, ob der ihm gegebene String ein Gemüse ist und nicht beispielsweise eine gleichnamige Rezeptzutat. Homonyme behandelt der Agent nicht.
- Der Agent extrahiert die gewünschte Information. Das zeitaufwändige Überfliegen von Webseiten auf der Suche nach relevanter Information wird dem Benutzer abgenommen.
- Der Agent sucht bei Bedarf die Informationen auf unterschiedlichen Seiten und stellt sie für den Benutzer zusammen. Der Anwender braucht nur noch eine einzige Anfrage zu stellen.
- Der Agent ist nicht nur auf statische Webseiten angewiesen, sondern kann auch Informationen aus Web Services erschließen.
- Der Agent ist in der Lage, Informationen, die auf der Webseite nicht explizit genannt sind, durch Inferenzprozesse zu erschließen.

Die Informationssuche im Internet wird deutlich verbessert, wenn nicht nur auf syntaktischer Ebene gearbeitet wird, sondern wenn die Suche auf einer formal definierten Semantik basiert. Die Technologien des Semantic Web bieten dabei eine vielversprechende Perspektive auf das Information Retrieval der Zukunft.

Es stellt sich nun die Frage, wie lange das Semantic Web noch auf sich warten lässt. Im W3C wird diese Frage meist damit beantwortet, dass das Semantic Web bereits in einer sehr rudimentären Form existiere. Dafür, dass die Idee erst um die Jahrtausendwende entstanden ist, hat sich erstaunlich viel getan: Seit 2004 gibt es eine solide Spezifikation für RDF, die quasi Standard geworden ist. Es gibt Programmierumgebungen für RDF in diversen Programmiersprachen wie C, C++, Python, Java, PHP und einigen mehr. RDF lässt sich in unterschiedlichen Formaten serialisieren wie XML, N3, Turtle etc. Ferner existieren einige Werkzeuge zur Bearbeitung von RDF in unterschiedlicher Qualität. Durch zahlreiche Veröffentlichungen und Tutorials kann man sich leicht Einblick in die Materie verschaffen. Da die Technologien aber noch relativ jung sind, muss man in allen Bereichen noch mit Bugs und Ungereimtheiten rechnen, die den Anwender viel Zeit und Geduld kosten. Auch

wenn es laut Herman (2007b, S. 5) geschätzte 10^7 Semantic Web-Dokumente gibt, wird eines der größten Probleme sein, das Internet in seinem Ausmaß zu annotieren und maschinenlesbar zu machen. Denn nur wenn ein ausreichender Pool an RDF-Dokumenten vorliegt, kann ein Agent effektiv in dem Semantic Web operieren.
Auch für OWL gibt es seit 2004 eine tragfähige Spezifikation, die die Benutzergemeinschaft einigermaßen als Standard anerkennt. Mit dem geschichteten Subtypen-Modell von OWL Full, OWL-DL und OWL Lite versuchen die Entwickler, die Balance zwischen Ausdrucksstärke und Implementierbarkeit bei unterschiedlichen Anwendungsbedürfnissen zu halten. Auch für OWL gibt es eine ganze Reihe von Entwicklungen wie Editoren, Programmierumgebungen in Java, Prolog und anderen Sprachen, Reasonern und Validatoren.
Das ausgemachte Ziel des Semantic Web ist die gemeinsame Nutzung und Wiederverwertbarkeit von Ontologien. Daher existieren bereits einige Ontologien und Vokabulare – am bekanntesten sind wohl der Dublin Core, vCard, FOAF, um nur einige zu nennen, die öffentlich zur Verfügung stehen und auch Anwendern mit geringen Fachkenntnissen die Teilnahme im Semantic Web ermöglichen sollen. Weil das Erstellen von brauchbaren Ontologien sehr zeitaufwändig ist und viel Erfahrung des Entwicklers voraussetzt, sind solche Ressourcen von unschätzbarem Wert. Von Nachteil ist allerdings, dass bei komplexeren Ontologien wie der Time-Ontologie gewisse Fachkenntnisse unumgänglich sind, weil man, ohne die Ontologie zu verstehen, nicht viel mit ihr ausrichten kann. Daher wird die Arbeit im und mit dem Semantic Web wohl doch ein gewisses Maß an Know-How erfordern und deutlich komplizierter bleiben als der Umgang mit dem bestehenden Internet.
Neben den hier beschriebenen Technologien, deren Entwicklung zwar noch nicht abgeschlossen ist, die aber immerhin schon funktionieren, gibt es noch einige kaum angetastete Bereiche im Semantic Web. So fehlen z.B. noch Sprachen für die Kommunikation von Agenten untereinander. Auch die Umsetzung der obersten Schichten der Semantic Web-Architektur wie *Proof* und *Trust* (siehe Abbildung 2.1, Seite 6) ist noch nicht geklärt.
Alles in Allem müssten neben den noch fehlenden Komponenten deutlich mehr Webseiten annotiert sein, deutlich mehr Ontologien zur Verfügung stehen und deutlich mehr Agenten implementiert sein, bis das Semantic Web zu seinem vollen Potential ausgereift ist. Besonders in unkommerziellen Wissensgebieten wie z.B. im Freizeit-Bereich gibt es bisher noch kaum Ansätze, weil vermutlich einerseits der Aufwand gescheut wird und andererseits kein unmittelbarer Nutzen auf der Hand liegt. Anders ist es bei kommerziellen Web-Anbietern, für die es aus geschäftlichen Gründen durchaus relevant sein kann, wenn Software-Agenten auf ihren Seiten agieren können. Daher verwundert es nicht, dass die Entwicklung des Semantic Web im kommerziellen Bereich schon am weitesten fortgeschritten ist.

Literaturverzeichnis

Altova: Was ist das Semantic Web? ⟨URL: http://www.altova.com/de/semantic_web.html⟩ – Zugriff am 11.08.2007

Antoniou, Grigoris/Franconi, Enrico/Harmelen, Frank van (2005): Introduction to Semantic Web Ontology Languages. In **Eisinger, Norbert/Małuszyński, Jan (Hrsg.):** Reasoning Web, Proceedings of the Summer School, Malta, 2005. Berlin ⟨URL: http://www.cs.vu.nl/~frankh/postscript/REWERSE05.pdf⟩ – Zugriff am 11.08.2007

Antoniou, Grigoris/Harmelen, Frank van (2004): A Semantic Web Primer. Cambridge

Baader, Franz (2003): The Description Logic Handbook, Theory, Implementation, and Applications. Cambridge

Berners-Lee, Tim (2006): Artificial Intelligence and the Semantic Web. ⟨URL: http://www.w3.org/2006/Talks/0718-aaai-tbl/Overview.html\#(1)⟩ – Zugriff am 15.08.2007

Berners-Lee, Tim/Hendler, James/Lassila, Ora (2001): The Semantic Web. Scientific American, 284(5), 34–43

Bijan, Parsia/Evren, Sirin (2004): Pellet: An OWL DL Reasoner. ⟨URL: http://iswc2004.semanticweb.org/posters/PID-ZWSCSLQK-1090286232.pdf⟩ – Zugriff am 11.08.2007

Birkenbihl, Klaus (2006): Standards für das Semantic Web. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 73–88

Blaumauer, Andreas/Pellegrini, Tassilo (2006): Semantic Web und Semantische Technologien: Zentrale Begriffe und Unterscheidungen. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 9–25

Bray, Tim (2000): Extensible Markup Language (XML) 1.0. ⟨URL: http://edition-w3c.de/TR/2000/REC-xml-20001006/⟩ – Zugriff am 11.08.2007

Bray, Tim/Hollander, Dave/Layman, Andrew et al. (2006): Namespaces in XML 1.0, W3C Recommendation. ⟨URL: `http://www.w3.org/TR/xml-names`⟩ – Zugriff am 15.08.2007

Bryson, Joanna/Martin, David et al. (2003): Agent-Based Composite Services in DAML-S: the Behavior-Oriented Design of an Intelligent Semantic Web. In **Zhong, Ning/Liu, Jiming/yao, yiyu (Hrsg.):** Web Intelligence. Berlin and Heidelberg, 37–58

Daconta, Michael/Obrst, Leo/Smith, Kevin (2003): The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management. New York

DAML: DAML Ontology Library. ⟨URL: `http://www.daml.org/ontologies/`⟩ – Zugriff am 13.08.2007

DAML-Time: Homepage. ⟨URL: `http://www.cs.rochester.edu/~ferguson/daml/`⟩ – Zugriff am 13.08.2007

Eckstein, Rainer/Eckstein, Silke (2004): XML und Datenmodellierung, XML-Schema und RDF zur Modellierung von Daten und Metadaten einsetzen. Heidelberg

Granitzer, Michael (2006): Statistische Verfahren der Textanalyse. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 437–452

Herman, Ivan (2007a): Tutorial on Semantic Web. ⟨URL: `http://www.w3.org/People/Ivan/CorePresentations/RDFTutorial/`⟩ – Zugriff am 11.08.2007

Herman, Ivan (2007b): State of the Semantic Web. ⟨URL: `http://www.w3.org/2007/Talks/0403-Tampere-IH/Slides.pdf`⟩ – Zugriff am 14.08.2007

Hess, Thomas (2006): Kombination semantischer Web Services auf Basis einer Domänenontologie am Beispiel diskographischer Informationen. Diplomarbeit, Fachhochschule Köln, ⟨URL: `http://mims02.gm.fh-koeln.de/miwebseite/cp/upload/cp\_projekte\_file\_paper\_12.pdf`⟩ – Zugriff am 11.08.2007

Hjelm, Johan (2001): Creating the Semantic Web with RDF. New York

Hobbs, Jerry/Pan, Feng (2006): Time Ontology in OWL. ⟨URL: `http://www.w3.org/2001/sw/BestPractices/OEP/Time-Ontology`⟩ – Zugriff am 13.6.07

Horridge, Matthew/Knublauch, Holger/Rector, Alan et al. (2004): A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools. ⟨URL: `http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf`⟩ – Zugriff am 11.08.2007

Kalyanpur, Aditya/Parsia, Bijan/Sirin, Evren et al. (2005): Swoop: A 'Web' Ontology Editing Browser. Journal of Web Semantics, 4(2) ⟨URL: `http://www.mindswap.org/papers/SwoopJWS\_Revised.pdf`⟩ – Zugriff am 11.08.2007

Kreuter, Marie-Luise (2000): Der Biogarten. München

Kunder, Maurice de: WorldWideWebSize, Daily estimated size of the World Wide Web. ⟨URL: `http://www.worldwidewebsize.com/`⟩ – Zugriff am 21.08.2007

Lockemann, Peter/Kirn, Stefan/Herzog, Otthein (2006): Agents. In **Kirn, Stefan/Herzog, Otthein/Spaniol, Otto (Hrsg.):** Multiagent Engeneering, Theory and Applications in Enterprises. Berlin and Heidelberg: Springer, 15–33

Martin, David/Burstein, Mark et al. (2004): OWL-S: Semantic Markup for Web Services. ⟨URL: `http://www.w3.org/Submission/OWL-S/`⟩ – Zugriff am 11.08.2007

May, Wolfgang (2006): Reasoning im und für das Semantic Web. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 485–503

Noy, Natalya/McGuinness, Deborah (2001): Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory Technical Report, KSL-01-05 ⟨URL: `http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf`⟩ – Zugriff am 11.08.2007

Ontolingua: Ontolingua Ontology Library. ⟨URL: `http://www.ksl.stanford.edu/software/ontolingua/`⟩ – Zugriff am 13.08.2007

OntoMat: Homepage. ⟨URL: `http://annotation.semanticweb.org/ontomat/index.html`⟩ – Zugriff am 11.08.2007

OntoMat: Tutorial. ⟨URL: `http://annotation.semanticweb.org/ontomat/tutorial.html`⟩ – Zugriff am 11.08.2007

Pan, Feng/Hobbs, Jerry (2004): Time in OWL-S. In AAAI Spring Symposium on Semantic Web Services. Stanford University ⟨URL: `http://www.isi.edu/~pan/time/pub/pan-hobbs-AAAI-SSS04.pdf`⟩ – Zugriff am 11.08.2007, 29–36

Pellet: Homepage. ⟨URL: `http://pellet.owldl.com/`⟩ – Zugriff am 01.07.2007

Polleres Axel, Lausen, Holger Lara Ruben (2006): Semantische Beschreibungen von Web Services. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 505–524

Prud'hommeaux, Eric/Seaborne, Andy (2007): SPARQL Query Language for RDF, W3C Candidate Recommendation. ⟨URL: `http://www.w3.org/TR/rdf-sparql-query/`⟩ – Zugriff am 24.08.2007

RDFLib: Homepage. ⟨URL: `http://rdflib.net/`⟩ – Zugriff am 11.08.2007

Reif, Gerald (2006): Semantische Annotation. In **Pellegrini, Tassilo (Hrsg.):** Semantic Web, Wege zur vernetzten Wissensgesellschaft. Berlin and Heidelberg, 405–418

Russell, Stuart/Norvig, Peter (2004): Künstliche Intelligenz, Ein moderner Ansatz. 2. Auflage. München

SMORE: Homepage. ⟨URL: `http://www.mindswap.org/2005/SMORE/`⟩ – Zugriff am 11.08.2007

Swoogle: Homepage. ⟨URL: `http://swoogle.umbc.edu/`⟩ – Zugriff am 11.08.2007

SWOOP: Homepage. ⟨URL: `SWOOP-Homepage:http://www.mindswap.org/2004/SWOOP/`⟩ – Zugriff am 11.08.2007

W3C: Frequently asked questions about RDF, How do I put some RDF into my HTML Page. ⟨URL: `http://www.w3.org/RDF/FAQ.html\#How`⟩ – Zugriff am 11.08.2007

W3C: Resource Description Framework (RDF). ⟨URL: `http://www.w3.org/RDF/`⟩ – Zugriff am 15.08.2007

W3C: Validation Service. ⟨URL: `http://www.w3.org/RDF/Validator/ARPServlet`⟩ – Zugriff am 01.07.2007

WonderWeb: OWL Ontology Validator. ⟨URL: `http://www.mygrid.org.uk/OWL/Validator`⟩ – Zugriff am 01.07.2007