Fakultät Umwelt und Technik
Bereich Informatik

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker (FH)

Integration IT - Embedded Systems: Wissensbasierte Kommunikation in verteilten Systemen mit der Web Ontology Language (OWL)

Angefertigt am
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur
Leibniz Universität Hannover

Betreut durch
Prof. Dr.-Ing. Ralph Welge
Prof. Dr. rer. nat. Dipl.-Inform. Helmut Faasch

Gideon Zenz
8. September 2006

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit einer Methodik, wissensbasierte Kommunikation in einem verteilten System zu realisieren. Als Szenario dient der *Organic Room*, ein Raum mit intelligenter Haustechnik, der sich möglichst eigenständig an die Bedürfnisse des Menschen anpasst und sich dabei energieeffizient verhält.

Die Ontologie ermöglicht es Geräten, Wissen über sich und ihre Fähigkeiten abzuspeichern und zu verarbeiten. Dabei ist dieses Wissen verknüpft und erlaubt so ein assoziatives Verständnis.

Es wird eine für den *Organic Room* passende Ontologie und eine für eingebettete Systeme geeignete Software vorgestellt. Sie kann die Wissen kommunizieren kann und es so dem Menschen ermöglicht, über eine zentrale Plattform dem Raum seine Bedürfnisse zu kommunizieren. Dieser reagiert darauf angemessen, in dem er für die Aktivität des Menschen passende Geräte identifiziert und entsprechend parametriert. Die transparente Weise, in der das System arbeitet, macht es sowohl effektiv als auch benutzerfreundlich.

Schlagwörter

Ontologie, eingebettete Systeme, Energieeffizienz, Nachhaltigkeit, Haustechnik, Wissensdatenbank

Abstract

This thesis describes a methodology that seeks to implement knowledge-based communication in a decentralized system. The example employed is an "organic room" which automatically adapts to the needs of its users in an energy-efficient way and based on intelligent home appliances. These appliances possess an "ontology", meaning that they have a form of self-knowledge such as e.g. an awareness of their capabilities. This knowledge is stored in the form of an associational network and enables an associational form of knowing.

The thesis outlines an ontology that is suited for this organic room as well as software that is appropriate for use in embedded systems, and which can communicate knowledge in a way that enables humans to use a central device to communicate their needs. With this technology, the organic room is capable to meet the needs of its users by automatically identifying and configuring all relevant appliances. The transparent fashion in which the system operates makes it both effective and user friendly.

Keywords

ontology, embedded systems, energy efficiency, sustainable, intelligent home, knowledgebase

Inhaltsverzeichnis

1. Einführung und Motivation

1.1 Motivation

Energie war für den Menschen schon immer ein kostbares Gut. Diese Erkenntnis ist aufgrund der technologischen Entwicklung in den vergangenen Jahrzehnten recht in den Hintergrund gerückt, wird aber langsam wieder bewusst wahrgenommen, was nicht zuletzt an steigenden Preisen aufgrund von Rohstoffknappheit liegt.

Ein wesentlicher Teil der weltweit verbrauchten Energie ist Strom. Gerade dieser Teil verzeichnet einen rasanten Anstieg. So ist der Verbrauch von ca. 439 Megatonnen Öleinheiten von 1973 auf fast 1175 Megatonnen im Jahre 2003 angewachsen. Bemerkenswert hierbei ist, dass der Anteil der Industrie am Verbrauch von 51,3% auf 42,2% sank, während der Anteil aus Agrarwirtschaft, öffentlichen Einrichtungen und Gewerbe und Haushalt von 46,3% auf 56% stieg. Der Anstieg findet seine Ursache also vor allem bei Kleinverbrauchern. [IEA 05]

Dies bedeutet, dass die Menschheit sich in wachsendem Maße mit dieser Problematik und ihren ökologischen Folgen beschäftigen muss. Neben dem Erschließen von regenerativen Energiequellen ist vor allem einen Mentalitätswandel notwendig. Energie muss wieder als etwas Kostbares betrachtet werden, was nicht verschwendet werden darf. Doch ist das gar nicht so einfach, denn vielfach ist nicht durchschaubar, wo wer überhaupt wie viel Energie verbraucht.

Der bewusste und umsichtige Umgang mit natürlichen Ressourcen wird unter dem Begriff der Nachhaltigkeit (engl. Sustainable) zusammengefasst. Die „World Commission on Environment and Development“ definierte 1987 „Sustainable Development“ als:

> „development that meets the needs of the present without compromising the ability of future generations to meet their own needs“ [WCOED 87]

Entwicklungen sollen also die heutigen Bedürfnisse befriedigen, ohne dabei zukünftige Generationen zu Hindern, ihre zu befriedigen. Längst spielt nachhaltiges Handeln auch in der Wirtschaft eine Rolle. Als Beispiel sei Toshiba genannt. In ihrer „Umwelt Vision 2010“ setzte sich Toshiba das Ziel, die Umwelteffizienz der Geräte im Zeitraum von 2000 bis 2010 um 120% zu steigern.

1.2 Szenario

Ziel im Rahmen des Forschungsprojektes **VAUST** (Verteilte autonome Systeme und Technologien) ist es, Installationstechnik für Wohn- und Zweckgebäude zu entwickeln, die den Energieverbrauch in Korrelation zum individuellen Handeln analysieren und damit den Menschen beim effizienten Umgang mit dieser Energie unterstützen soll.

Dies soll möglichst ohne Komforteinbußen für den Nutzer geschehen. Ein derart ausgestattetes Haus nennen wir *Organic House*, einen Raum in diesem Haus *Organic Room*.

Die Problematik hierbei liegt nicht zuletzt in der dramatisch gestiegenen Anzahl elektrischer Verbraucher. Standen gestern in einem Büro eine Lampe, ein Tisch, evtl. eine elektrische Schreibmaschine und ein Telefon, ist es heute mindestens mit einem PC ausgestattet. Die meisten werden über ein Handy verfügen, das geladen werden muss, manche einen PDA, etc.

Diese wachsende Anzahl von elektrischen Geräten macht es für den Menschen sehr unübersichtlich. Oft ist auch nicht klar, dass bereits ein eingestecktes aber nicht benutztes Ladegerät Energie verbraucht. Hier ist eine bessere Übersichtlichkeit der vorhandenen Verbraucher sowie bessere und zentrale Organisationsmöglichkeiten von Nöten.

1.3 Ziel

Um den Menschen beim energieeffizienten Handeln unterstützen zu können, benötigt ein Gerät Wissen über sich selbst und seine Fähigkeiten, aber auch Wissen über seine Umgebung und wie es mit dieser zusammenhängt. Wann ein Gerät benötigt wird, hängt immer von seiner Umgebung ab. Von daher muss ein Gerät in der Lage sein, seine Umgebung zu erkennen und zu interpretieren.

Um die gewünschte Funktionalität zu erreichen, bedienen wir uns der Analogie zum Menschen. Jeder Mensch weiß – in gewissem Rahmen – wer er ist und welche Aufgaben er hat. Dieses Wissen kann er mit anderen kommunizieren. Indem er das neue Wissen mit dem bereits Vorhandenen logisch verknüpft (assoziiert), versteht er die Zusammenhänge. Somit kann er insbesondere auch seine eigene Position und Aufgabe bestimmen, anpassen und – basierend auf diesem Wissen – intelligente Entscheidungen treffen. Merkt er bei dem Vorgang der Verknüpfung, dass ihm zum Verständnis notwendiges Wissen fehlt, das er nicht assoziativ aus dem bisherigen ableiten kann (Nachdenken), so kann er durch Nachfragen diese Lücke schließen.

Dieses Paradigma macht sich die informatische Ontologie zunutze. Sie ist ein formales System, das Konzepte und Relationen sowie Inferenz- und Integritätsregeln definiert. Hiermit lassen sich also nicht nur Information speichern, sondern auch Zusammenhänge erkennen und durch Anwendung von Regeln neue Zusammenhänge bilden.

Mit dem Ansatz der *Web Ontology Language* des *World Wide Web Consortium* soll ein für *Embbeded Systems*[1] taugliches, ontologisches System entwickelt werden. Mit dem Modell des *Organic House* ausgestattet, sollen Geräte so in der Lage sein, ihre Umgebung assoziativ zu erkennen und zu den Aktivitäten des Menschen passende Aktionen automatisch ermitteln zu können.

Besonderes Augenmerk liegt hier neben der Energieeffizienz darin, dem Menschen eine natürlichere und zentrale Steuerungsmöglichkeit seiner elektronischen Umgebung zu bieten. Durch ihr assoziatives Verständnis ermöglicht das technische System dem Menschen, mit ihm auf viel natürlichere Art und Weise zu kommunizieren als es heutzutage möglich ist. Weiterhin ist es wichtig, den Menschen weder zu bevormunden noch zu überfordern. Nicht jeder will ständig alle Gerät im Detail steuern, solange diese etwas tun, was im Rahmen seiner Erwartungen liegt und zu der Aktivität passt, die er gerade ausführen möchte. Daher sollten Geräte in der Lage sein, zu gewünschten Aktionen des Menschen passende Aktionen ihrerseits zu ermitteln und diese dem Menschen anzubieten. Weiterhin wäre es wünschenswert, über eine zentrale, möglichst tragbare Steuereinrichtung zu verfügen.

So soll der Wohnraum in der Lage sein, aus einem Wunsch bestmöglich eigenständig Handlungen abzuleiten. Wenn beispielsweise der Mensch konzentriert arbeiten möchte, so wäre es sinnvoll automatisch für eine passende Raumtemperatur, passende Beleuchtung und Ruhe zu sorgen. Will er entspannen, so wäre eine entsprechend andere Beleuchtung, evtl. anderer Teile des Raumes, wünschenswert, passende Musik bzw. Filme. Zum Schlafen wäre wieder eine andere Raumtemperatur, Beleuchtung oder Musik passend.

[1] stromsparende Kleinstgeräte

Die Ontologie ermöglicht es dem System Haus, aus einem einfachen Wunsch (z.B. „Brief schreiben“ etc.) autonom Folgerungen für das gesamte System abzuleiten. So muss eben nicht mehr jedes Gerät manuell wie gewünscht eingestellt werden, die einfache Äußerung des Wunsches veranlasst dieses automatisch.

Natürlich ergibt sich auch hier Bedarf zum Nachfragen, damit sich System und Mensch auch wirklich verstehen. Jeder wird hier seine eigenen Assoziationen zu einer gewünschten Tätigkeit haben, worauf sich das System entsprechend einstellen muss. Auch zwischen den Systemen wird es Bedarf für Informationsaustausch geben, damit sie zu sinnvollen Aktionsvorschlägen kommen können.

1.4 Konventionen

In dieser Arbeit werden *Fachbegriffe* und *Schlüsselworte* grundsätzlich *kursiv* gedruckt. **Beispiele** werden grundsätzlich **fett** gedruckt. Beispiel:

> „Im Beispiel ist die *Klasse* **Freizeit** *Subklasse* von **Component“**

Bedeutet, dass mit „*Klasse*“ und „*Subklasse*“ Sprachelemente gemeint sind. **Freizeit** und **Component** sind dabei konkrete Beispiele für diese Sprachelemente.

Des weiteren werden **Produktnamen** auch generell fett gedruckt.

1.5 Überblick

In den folgenden Kapiteln 2 und 3 wird zunächst die *Web Ontology Language OWL* und die Techniken, auf denen sie basiert eingeführt. Daraufhin erläutert Kapitel 4 Tools zur Erstellung von Ontologien.

Darauf basierend definiert Kapitel 5 eine Ontologie für den *Organic Room* und Kapitel 6 stellt schließlich die Software für den *Organic Room* inklusive der neu entwickelten **Embedded Ontology Engine** dar.

Im Anhang befindet sich die JavaDoc Dokumentation der Engine, die formale Definition von *OWL* zur Referenz sowie einige Klassen- und Aktivitätsdiagramme mit detaillierter Beschreibung der Engine und der *Organic Room* Software.

2. Stand der Technik

In diesem Kapitel wird die *Web Ontology Language OWL* vorgestellt, sowie alle Techniken, auf denen sie basiert.

In Kapitel 2.1 wird kurz auf die Ursprünge der Ontologie eingegangen. Danach werden die Basistechniken *HTML*, *XML*, *RDF* und *RDFS* kurz vorgestellt. Eine detaillierte Einführung in OWL folgt in Kapitel 3. Alle Techniken werden anhand des Beispiels *Organic Room* erläutert.

Das Kapitel endet mit einem kurzen Überblick.

2.1 Ursprünge der Ontologie

Die Ontologie, aus dem Griechischen óv als Partizip zu ειναι - „sein“ und aus λόγος - „Lehre“, „Wort“, ist ursprünglich eine philosophische Disziplin. Sie beschäftigt sich mit dem Sein als Abstraktum. Es geht hier also nicht um spezielle Fragen wie „Was ist der Mensch“, sondern darum, was das Sein als solches bedeutet und welche Eigenschaften es besitzt – ergo die Philosophie des Seins.

Diesen grundsätzlichen Ansatz hat die künstliche Intelligenzforschung aufgegriffen, um eine Methodik zu entwickeln, einen „Weltausschnitt“ zu konzeptualisieren und diesen formalisiert (damit maschinenverständlich) abzulegen und zu bearbeiten.

Die bekannteste Definition hierzu liefert [Gruber 93]:

> „An **ontology** is an explicit specification of a conceptualization.”

Eine Ontologie konzeptualisiert also das Sein, und zwar mittels einer expliziten bzw. formalen Spezifikation, so dass sie maschinenverständlich ist. Das „Sein“ ist hierbei alles, was darstellbar an sich ist. Insbesondere wird das Wissen über einen bestimmten Bereich („domain of interest“) formalisiert dargestellt. Es werden also alle Objekte, die zu diesem Bereich gehört, dargestellt und die Beziehungen zwischen ihnen.

Diesen Ansatz benutzen Tim Berners-Lee et al. um das sogenannte *Semantic Web* zu propagieren. Sie definieren es wie folgt:

> “The SemanticWeb is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.” [Lee et al. 01]

Hierbei spielt die Ontologie eine zentrale Rolle, die genau diese Schnittstelle einnimmt: Sie ermöglicht es Menschen und Computern, sich besser zu verstehen und zusammenzuarbeiten.

In diesem Rahmen wurde in den vergangenen Jahren die Spezifizierung von Standards und Entwicklung von Tools vor allem durch das *W3C* vorangetrieben.

2.2 Basistechnologien

Das *Semantic Web* basiert auf einer Reihe von Technologien des Webs, die im Folgenden kurz anhand des Beispiels des *Organic Rooms* eingeführt werden. Für jeden Schritt werden die Vorteile der jeweils weiteren Technologie für die Zielsetzung, Mensch und Maschine kooperieren zu lassen, im Vergleich zur Vorhergehenden aufgezeigt.

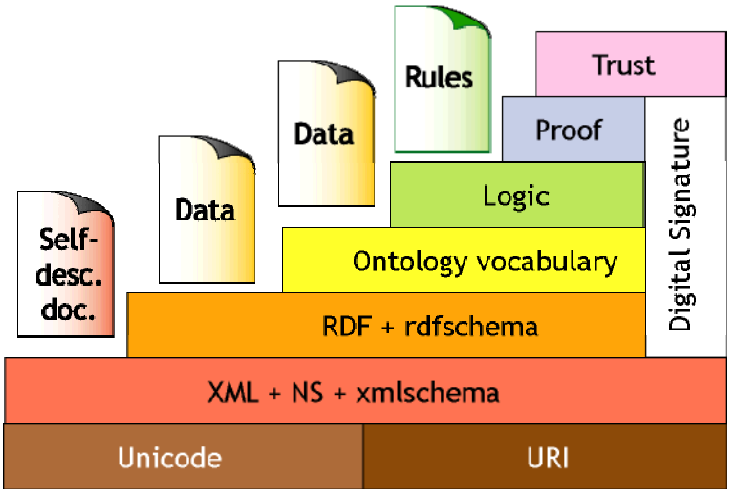


Abbildung 1: Der Semantic Web Tower

Aus [Lee 00]

Die Abbildung zeigt den Semantic Web Tower, der eine Schichtung von Protokollen darstellt. Mit zunehmender Höhe steigt die Mächtigkeit der Protokolle, Aussagen zu formulieren. Als Basis dient *XML*, siehe Kapitel 2.2.2, mit dem man strukturierte, maschinenlesbare Dokumente erstellen kann. Es benutzt Unicode als Zeichensatz und sogenannte *URIs, Uniform Resource Identifiers* zur Referenzierung von anderen Webressourcen[2].

Das *Resource Description Framework RDF* ist eine Sprache zur Beschreibung von diesen Webressourcen. Es ist grafisch oder mithilfe von *XML* darstellbar. Näheres hierzu in Kapitel 2.2.3.

Das *Resource Description Framework Schema (RDFS)* erweitert *RDF* um eine Semantik. Es definiert Klassen, Eigenschaften und erlaubt Restriktionen auf diese und ermöglicht hiermit erstmals inhaltliche Aussagen festzuhalten. Näheres hierzu in Kapitel 2.3.

Eine Erweiterung hiervon bildet die *Ontology Vocabulary* Schicht, definiert durch die *Web Ontology Language OWL*. Sie erlaubt komplexere Beziehungen zwischen Webobjekten und erweitert *RDFS*. Näheres hierzu in Kapitel 3.

Den Abschluss bildet der *Logic*, der *Proof* und der *Trust Layer*. Ersterer erweitert die Ontologie um die Möglichkeit, anwendungsspezifische Regeln zu bilden. Der *Proof-Layer* kann hieraus Schlussfolgerungen ableiten sowie Beweise abbilden.

Der *Trust-Layer* bietet schließlich Sicherheitskonzepte, um die Integrität und Authentizität der Daten über digitale Signaturen sicherzustellen. Diese beiden Ebenen gehen über das für diese Arbeit benötigte hinaus und werden deshalb nicht weiter detailliert.

2.2.1 HTML

Fundament des heutigen *World Wide Webs (WWW)* ist die *Hypertext Markup Language HTML*. Sie wurde 1989 von Tim Berners-Lee am CERN in Genf entwickelt und später von dem dafür gegründeten *World Wide Web Consortium* W3C standardisiert[3]. Diese Sprache ist eine Art Seitenbeschreibungssprache und legt mithilfe von *Tags* fest, wie Texte von einem Browser anzuzeigen sind. So ist es für einen Benutzer in dem *Organic Room* interessant, welche Geräte in dem Raum

[2] Ein Beispiel für eine URI ist ein normaler Weblink, also Beispielsweise http://www.uni-lueneburg.de
[3] Die Spezifikation ist unter http://www.w3.org/TR/html4/ zu finden.

vorhanden sind und welche Fähigkeiten sie besitzen. In *HTML* sähe dieses beispielsweise so aus:

```
01: <h1> Geräte im <em>Organic Room</em>
02: <h2> Fernseher
03: <h5> Filme
04: <h2> Lampe </h2>
05: <h5> Lichtquelle </h5>
06: <h2> Radio </h2>
07: <h5> Musik </h5>
```

Listing 1: Organic Room Beispiel in HTML

Die Ausgabe des Browsers ist für den Menschen leicht interpretierbar, durch die *HTML* Anweisungen wurde der Inhalt klar gegliedert dargestellt:

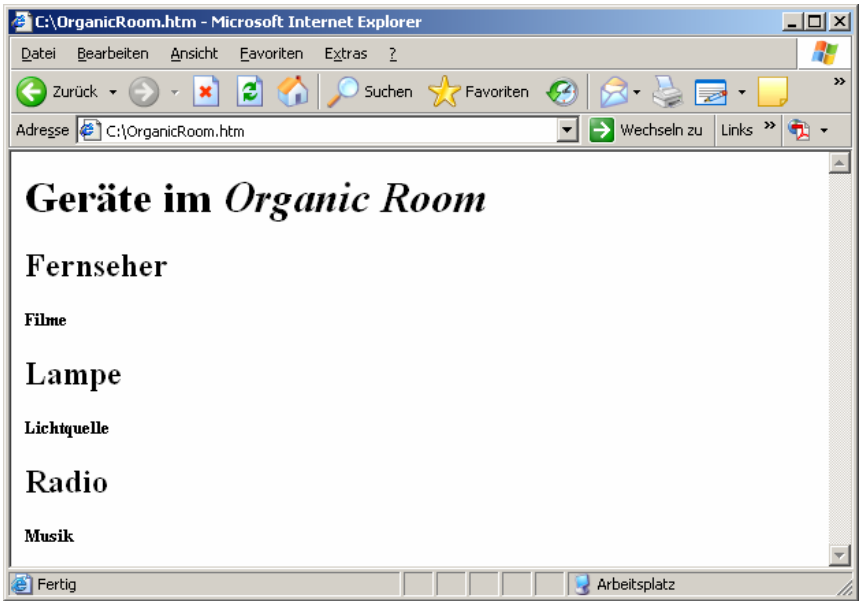


Abbildung 2: Organic Room Beispiel in HTML

Wie anhand der Ausgabe zu sehen ist, sorgen die *<h>-Tags* für eine Hervorhebung der betreffenden Textstellen.

Problematischer dagegen ist es, die Informationen mittels eines Programmes zu interpretieren. Die Tags erleichtern zwar dem Menschen die Interpretation des Inhaltes, geben dem Computer aber keinen genauen Anhalt über Bedeutung und Zusammenhänge des Wissens.

So ist beispielsweise eine Anfrage nach Geräten für die Freizeit nicht so einfach zu beantworten. Wünschenswert wäre es, hier den Fernseher und das Radio zu finden.
Weiterhin zu sehen ist, dass *HTML* keine besonders strikte Syntaxprüfung beinhaltet. So wird der *Tag* `<h5>` in Zeile 3 nicht geschlossen, die Darstellung erfolgt jedoch trotzdem korrekt – zumindest in dem verwendeten Browser.

2.2.2 XML

Mit der *Extensible Markup Language XML* hat das W3C 1998 eine Metasprache zur Erstellung von Dokumenten verabschiedet, die sowohl für Maschinen als auch für Menschen les- und interpretierbar sein sollen. *XML*[4] selbst ist eine Teilmenge von *SGML*, der *Standard Generalized Markup Language*, ISO 8879:1986.

[4] Die Spezifikation ist unter http://www.w3.org/XML/ zu finden.

Der Vorteil wird leicht ersichtlich, wenn obiges *HTML* Beispiel mit *XML* wiedergegeben wird:

```
01: <?xml version="1.0" encoding="ISO-8859-1"?>
02: <!DOCTYPE autonomicunits SYSTEM "organicroom.dtd">
03: <autonomicunits name="Organic Room">
04:  <unit name="Fernseher" beschreibung="Filme"/>
05:  <unit name="Lampe" beschreibung="Lichtquelle"/>
06:  <unit name="Radio" beschreibung="Musik"/>
07: </autonomicunits>
```

Listing 2: Organic Room Beispiel in XML

Wie zu sehen ist, wird hier nicht wie eben das Aussehen per *Tags* vorgegeben, die in unterschiedlichen Textgrößen resultieren, sondern eine Syntaktik gebraucht, die zu dem *Organic Room* passt. Dieser wird in der sogenannten *Document Type Definition (DTD)* festgelegt, die den allgemeinen Aufbau eines Gerätes festlegt. Sie wird in Zeile 2 aus der Datei **organicroom.dtd** nachgeladen. Daraufhin wird der *OrganicRoom* in Zeile 3 samt Namen definiert. Darauf folgen die einzelnen Geräte (*units*) jeweils mit Namen und Beschreibung. Im Gegensatz zu *HTML* ist es auch hier unabdingbar, alle *Tags* ordnungsgemäß zu schließen, da ansonsten der *Parser*, der das Dokument verarbeitet, einen Fehler melden würde. Die Strukturbeschreibung sieht wie folgt aus:

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <!ATTLIST autonomicunits name CDATA #REQUIRED>
03: <!ELEMENT autonomicunits (unit+)>
04: <!ATTLIST unit
05:  name CDATA #REQUIRED
06:  beschreibung NMTOKEN #REQUIRED
07: >
08: <!ELEMENT unit EMPTY>
```

Listing 3: Organic Room Beispiel in XML (DTD)

Hiermit hat ein *Organic Room* also eine sehr konkrete Struktur inklusive verschiedener Restriktionen erhalten. So wird in Zeile 2 festgelegt, dass ein *Organic Room* einen Namen und in Zeile 3, dass er mindestens eine *unit* haben muss. Weiterhin in den Zeilen 5 und 6, dass ein *Unit* einen Namen und eine Beschreibung haben muss.

Hieraus geht also unmittelbar für die Maschine verständlich hervor, wie die vorher bedeutungslosen Wörter zusammenhängen und einzuordnen sind.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE autonomicunits (View Source for full doctype...)>
- <autonomicunits name="Organic Room">
    <unit name="Fernseher" beschreibung="Filme" />
    <unit name="Lampe" beschreibung="Lichtquelle" />
    <unit name="Radio" beschreibung="Musik" />
  </autonomicunits>
```

Abbildung 3: Organic Room Beispiel in XML

Allerdings wird durch *XML* weiterhin keine Semantik definiert. So ist der Zusammenhang beispielsweise zwischen Fernseher und Radio weiterhin ungeklärt.

2.2.3 RDF

Das *Resource Description Framework RDF* ist eine Sprache zur Beschreibung von Metadaten im Web. *RDF* bildet ein Grundstein des *Semantic Web* (vgl. Kapitel 2.2) und

wurde 2000 veröffentlicht[5]. Für *RDF* gibt es eine grafische Darstellung und eine textuelle mittels *XML*.

RDF besteht aus drei Kernkonzepten:

1. *Resources*:
   Sind alle Objekte, die durch *RDF* beschrieben werden können, d. h. durch *URIs* identifiziert werden können. Diese können sein:
   - Objekte, die über das Netz erreicht werden können, wie Webseiten, Bilder etc.
   - Objekte, die nicht über das Netz zu erreichen sind, wie Menschen, Bücher etc.
   - Abstrakte Konzepte wie Datenmodelle und –strukturen.

2. *Properties*:
   Verbinden Ressourcen miteinander. Beispiele im *OrganicRoom* sind **Name** oder **PowerConsumption**. Ersteres „verbindet" ein Gerät mit seiner Beschreibung, das zweite mit dem Wert, der seinen Energieverbrauch beziffert. Sie werden ebenfalls mittels *URIs* gebildet und sind dadurch insbesondere ebenso *Ressourcen*.

3. *Statements*:
   Verknüpfen eine *Property* mit einem konkreten Wert. Ein *Statement* besteht also aus Objekt, *Property* und Wert. Ein Objekt ist immer eine *Resource*, ein Attribut kann allerdings eine *Resource* oder ein *Literal*, also ein atomarer Wert, sein.

Betrachten wir die Aussage:

> Der Fernseher stellt Filme zur Verfügung und verbraucht 300 W.

So lassen sich daraus folgende *Statements* generieren:

> („Fernseher", http://organicroom.org/PowerConsumption, "300")
> („Fernseher", http://organicroom.org/has, "Filme")
> (http://organicroom.org/AutonomicUnit.rdfs#TV,
> http://organicroom.org/name, "Fernseher")

Die Attribute der ersten beiden *Statements* sind also *Literals*, das vom Letzten eine *Ressource*. Das letzte *Statement* formalisiert die Person selbst, d.h. macht sie als *RDF* Objekt unter „TV" verfügbar.

Aus diesen Statements generiert sich die folgende grafische Darstellung:

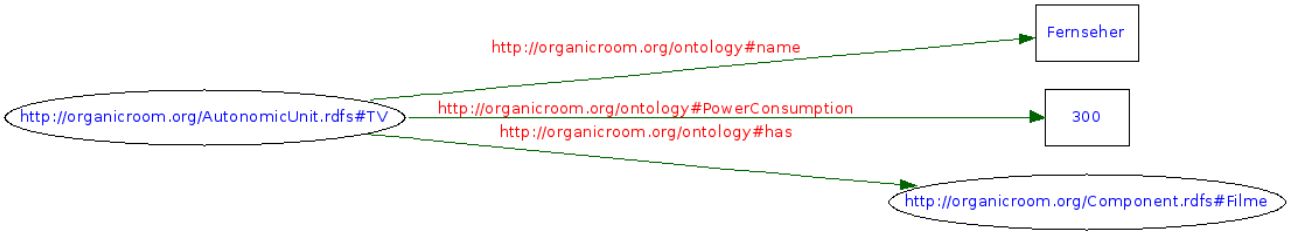


Abbildung 4: RDF Triples in grafischer Darstellung

(Hergestellt mit http://www.w3.org/RDF/Validator/)

[5] Die Spezifikation ist unter http://www.w3.org/RDF/ zu finden.

Die korrelierende textuelle Repräsentation ist:

```
01: <?xml version="1.0" encoding="ISO-8859-1"?>
02: <rdf:RDF
03: xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
04: xmlns:unit="http://organicroom.org/ontology#">
05: <rdf:Description
      rdf:about="http:// organicroom.org/AutonomicUnit.rdfs#TV">
06: <unit:name>Fernseher</unit:name>
07: <unit:PowerConsumption>300</unit: PowerConsumption>
08: <unit:has rdf:resource=
      "http://organicroom.org/Component.rdfs#Filme"/>
09: </rdf:Description>
10: </rdf:RDF>
```

Listing 4: RDF Triples in textueller Darstellung

In Zeile 5 erfolgt die Deklaration des Objektes Fernseher, entsprechend dem letzten Tripel im Beispiel. Darauf folgt das erste mit dem Namen, „Fernseher“ und danach das mit dem Energieverbrauch. Das letzte beschreibt die Komponente des Gerätes, also was es zur Verfügung stellt, nämlich Filme.

Im Vergleich zu *XML* lässt sich schon deutlich mehr über die Objekte aussagen. Jedoch sind Zusammenhänge zwischen den Geräten damit immer noch nicht ausgedrückt. *RDF* bietet hier die Möglichkeit der *Klassifizierung* mittels des Elements *rdf:type*. So könnte man Radio und Fernseher beide als Freizeitgeräte klassifizieren. Damit änderte sich das Beispiel für den Fernseher von oben folgendermaßen:

```
01: <rdf:Description
      rdf:about="http://domain.org/AutonomicUnit.rdfs#TV">
02:   <rdf:type rdf:resource="&unit;freizeit">
03:   ...
04: </rdf:Description>
05:
06: <rdf:Description
     rdf:about="http://domain.org/AutonomicUnit.rdfs#Radio">
07:   <rdf:type rdf:resource="&unit;freizeit">
08: ...
```

Listing 5: RDF Klassifizierung mittels rdf:type

Hiermit lässt sich die Frage nach Freizeitgeräten einfach beantworten, allerdings gehen dabei ihre speziellen Eigenarten verloren, nämlich dass das eine Musik (Radio), das andere Filme (Fernseher) zur Verfügung stellt. Ein Subklassenkonzept, mit dem sich Musik bzw. Film als Unterklasse von Freizeitgeräten deklarieren ließe, ist jedoch in *RDF* nicht vorgesehen.

2.3 RDF Schema

RDF Schema (RDFS), seinerseits die Kurzform für *RDF Vocabulary Description Language*, ist eine Erweiterung von *RDF* in *RDF*, d.h. es stellt im wesentlichen eine Bibliothek mit vordefinierten Deklarationen dar. Es wurde 2004 vom W3C vorgestellt[6]. *RDFS* ähnelt hierbei den aus der objektorientierten Programmierung bekannten Konzepten und erlaubt erstmals, Ontologien darzustellen. Es erlaubt Ressourcen zu klassifizieren und Eigenschaften (*Properties*) für diese Klassen sowie *Instanzen* dieser *Klassen* zu definieren.

[6] Die Spezifikation ist unter http://www.w3.org/TR/rdf-schema/ zu finden.

Das folgende Bild zeigt die Umsetzung des Beispiels in *RDFS*. Hierbei wird zwischen den *Klassen* selbst, also im Beispiel Geräten im Raum und ihren *Instanzen*, also den konkreten Geräten wie Radio selbst unterschieden. Die *Instanzen* der Klassen sind im *RDF Layer* als Kreise enthalten. Der *RDFS Layer* hingegen enthält die Struktur, die der *RDF Layer* benutzt. Hier sind *Klassen* als Kreise dargestellt und *Properties* als Rechtecke.

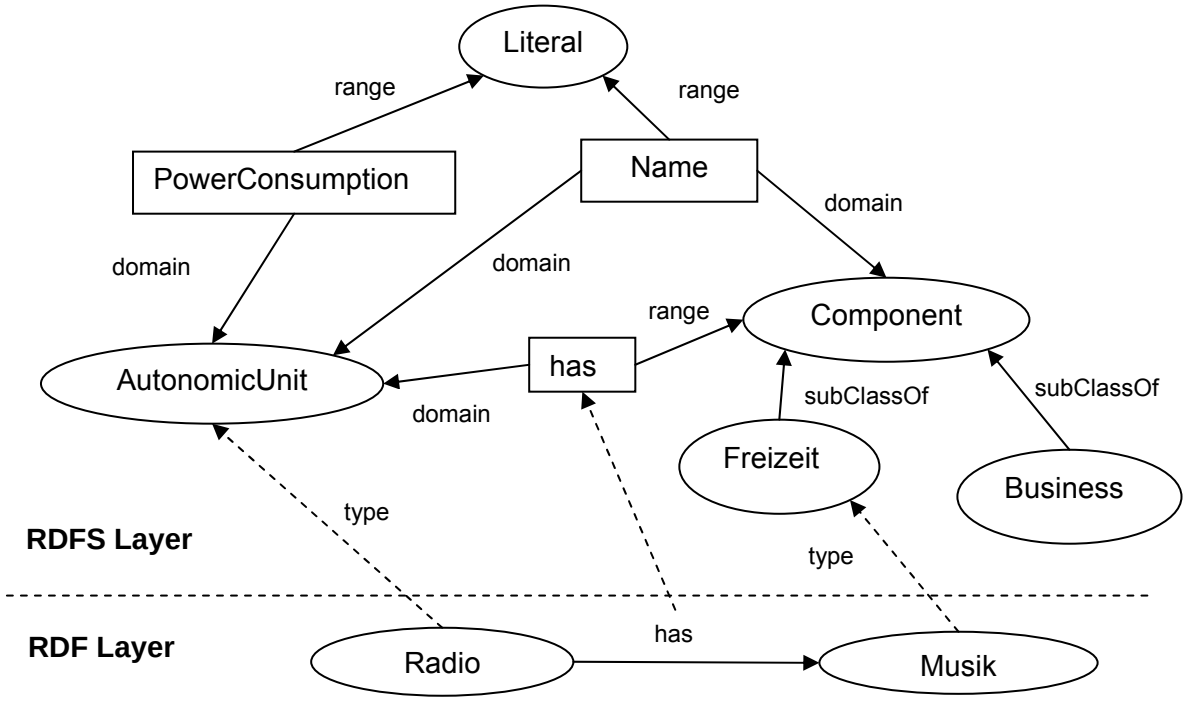


Abbildung 5: RDFS Klassenstruktur Organic Room

Hieraus lassen sich die wichtigsten Unterschiede zwischen *RDF* und *RDFS* entnehmen:

Klassen und Instanzen: *RDFS* definiert *Klassen*. *Klassen* haben Eigenschaften, genannt *Properties*. Mit *RDF* können nun *Instanzen* dieser Klassen definiert werden und Aussagen über diese gemacht werden, indem die entsprechenden *Properites* benutzt werden.

Hierarchien: Die Beschränkung von *rdf:type* wurde hier durch das Konzept von *Subklassen* erweitert. Im Beispiel ist die *Klasse* **Freizeit** *Subklasse* von **Component**, damit also auch eine **Component**, aber eine spezielle. Hiermit ist also die in Kapitel 2.2.3 erwähnte Unzulänglichkeit von *RDF* aufgehoben.

Restriktionen: *Properties* können jetzt beschränkt werden, indem ihnen ein Wertebereich (*range*) und einem Definitionsbereich (*domain*) spezifiziert werden. Mit ersterem wird festgelegt, auf welche Typen die *Property* verweisen kann, mit letzterem, in welchen *Klassen* sie Verwendung findet. So wird sichergestellt, dass in die *Property* **has** nur Objekte des Typs **Component** eingesetzt werden und keine anderen.

Die textuelle Repräsentation des Beispiels wäre:

```
01: <rdfs:Class rdf:about="AutonomicUnit">
02: </rdfs:Class>
03: <rdfs:Class rdf:about="Component">
04: </rdfs:Class>
05: <rdfs:Class rdf:about="Freizeit">
06: <rdfs:subClassOf rdf:resource="#Component"/>
07: </rdfs:Class>
08: <rdfs:Class rdf:about="Business ">
09: <rdfs:subClassOf rdf:resource="#Component "/>
10: </rdfs:Class>
11:
12: <rdf:Property rdf:about="PowerConsumption">
13: <rdfs:domain rdf:resource="#AutonomicUnit"/>
14: <rdfs:range rdf:resource="&rdf;Literal"/>
15: </rdf:Property>
16: <rdf:Property rdf:about="Name">
17: <rdfs:domain rdf:resource="#AutonomicUnit"/>
18: <rdfs:domain rdf:resource="#Component"/>
19: <rdfs:range rdf:resource="&rdf;Literal"/>
20: </rdf:Property>
21: <rdf:Property rdf:about="has">
22: <rdfs:domain rdf:resource="#AutonomicUnit"/>
23: <rdfs:range rdf:resource="#Component"/>
24: </rdf:Property>
```

Listing 6: RDFS Klassenstruktur Organic Room

In den Zeilen 1-10 werden zunächst die *Klassen* **AutonomicUnit**, **Component**, **Freizeit** und **Business** deklariert. Hierbei werden **Freizeit** und **Business** als *Subklassen* von **Component** gekennzeichnet.

Danach werden die *Properties* in den Zeilen 12-24 definiert, die diese Klassen gebrauchen. Die *Domain* sagt jeweils, in welcher *Klasse* diese *Property* enthalten ist, die *Range*, auf welche Klasse sie sich bezieht. Die Range kann auch ein Literal sein, wie z.B. bei **Name** oder **PowerConsumption**.

Die Eigenschaft der Restriktion ermöglicht erstmals auch eine neue Funktionalität, nämlich die inhaltliche Validierung. So kann eine Anwendung überprüfen, ob tatsächlich eine gültige Ontologie vorhanden ist, in der die *Ranges* und *Domains* der *Properties* auch überprüft werden. So wird im Beispiel sichergestellt, dass in die *Property* **has** nur Objekte des Typs **Component** eingesetzt werden und keine anderen.

2.4 Zusammenfassung

Im Kapitel 2 wurde definiert, was eine Ontologie ist und welche derzeitigen Ansätze es gibt, diese in der Informatik umzusetzen. Hierbei spielt das *Semantic Web* des W3C eine herausragende Rolle, weshalb die Basistechnologien, die auf die ontologische Variante, *OWL*, hinführen, im Detail am Beispiel eines vereinfachten *Organic Room* erläutert wurden. Hierbei zeigte sich, wie von einfacher Darstellungstechnik wie *HTML* (Kap 2.2.1), über erste strukturierte und maschinenverarbeitbare Darstellungen wie *XML* (Kap. 2.2.2) sich mittels *RDF* (Kap. 2.2.3) erstmals logische Zusammenhänge ausdrücken ließen. Schließlich wurde mit *RDFS* (Kap. 2.3) die erste Grundtechnik, Wissen semantisch und ontologisch zu speichern, erläutert.

Aufgrund der Wichtigkeit für diese Arbeit wird im Folgenden *OWL* ein eigenes Kapitel gewidmet.

3. Die Web Ontology Language OWL

Im Kapitel 2 wurde in die Vorläufertechnologien von *OWL* eingeführt. Im Folgenden soll nun detailliert auf *OWL* eingegangen werden. Abschließend folgt eine Bewertung der Eignung hinsichtlich des Szenarios *Organic Room* sowie eine Zusammenfassung und Literaturempfehlungen.

3.1 Einführung

OWL wurde von der W3C 2004 verabschiedet[7]. Wie in der Einführung in Kapitel 2.1 erläutert, erlauben ontologische Sprachen, explizite, formalisierte Konzeptualisierungen von Weltausschnitten auszudrücken. Daraus leiten sich die Bedingungen an eine Ontologiesprache ab. Sie benötigt:

- Eine wohldefinierte **Syntax**
- Eine formale **Semantik**
- Effiziente Unterstützung für ***Reasoner***
- Ausreichende **Ausdrucksmöglichkeiten**

Eine wohldefinierte **Syntax** ist eine Grundvoraussetzung, ohne die eine Sprache nicht maschinenverarbeitbar wäre.

Eine formale **Semantik** heißt, die Bedeutung von Wissen präzise zu beschreiben. Präzise bedeutet, dass die Semantik nicht auf subjektiver Intuition basiert oder Uneindeutigkeiten bezüglich der Interpretation aufweist. Dies erlaubt es, Schlussfolgerungen streng logisch aus dem Wissen zu ziehen. Dieser Vorgang wird ***Reasoning*** oder ***Inferenz*** genannt.

Das *Reasoning* ermöglicht folgendes festzustellen:

- **Klassenzugehörigkeit**: Wenn **x** eine *Instanz* der *Klasse* **C** ist, und **C** *Subklasse* von **D**, dann kann daraus abgeleitet werden, dass **x** auch *Instanz* von **D** ist.

- **Äquivalenz**: Wenn **A** und **B** identische Eigenschaften haben, so sind sie äquivalent. Gilt weiterhin, dass **B** zu **C** äquivalent ist, so folgt zusätzlich, dass **A** auch zu **C** äquivalent ist.

- **Konsistenz**: Es kann überprüft werden, ob die Ontologe gültig ist. Sei Beispielsweise **x** eine Instanz der *Klasse* **A** und **A Subklasse** sowohl von **B** als auch von **C**. Sind **C** und **B** als *disjunkt* angegeben, so folgt daraus eine **Inkonsistenz**, da **A** leer sein müsste, jedoch das Element **x** enthält.

- **Klassifikation**: Die Definition einer *Klasse* gibt an, welche *Properties* eine Instanz haben muss, um zu dieser *Klasse* gehören zu können. Erfüllt jetzt eine beliebige *Instanz* diese Anforderungen, dann folgt daraus, dass sie Mitglied dieser *Klasse* sein muss, auch wenn dies bisher nicht explizit angegeben war.

Die Hauptaufgaben des *Reasoning* sind also, die **Konsistenz** der Ontologie sicherzustellen, ungewollte Beziehungen zwischen *Klassen* herauszufinden sowie automatisch *Instanzen Klassen* zuzuordnen.

Mit *RDF* und *RDFS* lässt sich bereits einiges an ontologischem Wissen ausdrücken. So lässt sich das Wissen *klassifizieren*, in *Hierarchien* einteilen und mit *Eigenschaften*

[7] Die Spezifikation ist unter http://www.w3.org/TR/owl-semantics/ zu finden.

versehen, die wiederum *Restriktionen* unterworfen werden können. Allerdings fehlen *RDFS* noch eine Reihe von Ausdrucksmöglichkeiten, die wünschenswert wären. *OWL* ermöglicht z.B. auch folgendes:

- Lokalität der Restriktionen:
  Ranges gelten bei *RDFS* bei einer *Property* immer, d.h. bei allen *Klassen* in der *Domain* der *Property* gleichermaßen. So wäre es z.B. nicht möglich, eine *Subklasse* **RecreativeAutonomicUnit** zu definieren, bei der die *Property* **has** die *Restriktion* hat, nur Elemente der *Klasse* **Freizeit** referenzieren zu dürfen, ohne das dies für alle *Klassen* in der *Domain* von **has** der Fall wäre.

- Disjunkte Klassen:
  RDF lässt ohne weiteres mehrere *rdf:type* Tags zu, entsprechend können *Instanzen* auch immer zu mehreren *Klassen* gehören. Das ist jedoch nicht immer gewünscht, aber mit *RDF* nicht zu verhindern.

- Boolsche Verknüpfungen:
  Manchmal ist es ebenso wünschenswert, Restriktionen aus einer logischen Aussage zu bilden. So könnte z.B. eine *Klasse* **MultifunctionalUnit** definiert sein als alle Geräte, die sowohl in der *Klasse* **Freizeit** als auch der *Klasse* **Business** enthalten sind.

- Kardinalitäten:
  Es ist sinnig, festzulegen, dass ein **AutonomicUnit** mindestens eine **Component** haben muss, sonst wäre es ja funktionslos. Auch hierfür fehlt *RDFS* die Ausdrucksmöglichkeit.

- Beziehungen zwischen Eigenschaften:
  Manche Eigenschaften sind zueinander invers (*hatMitglied* und *istMitgliedVon* wäre hierzu ein Beispiel) oder sind *funktional*, können also immer nur einen eindeutigen Wert beinhalten oder verhalten sich transitiv. Transitiv wäre z.B. *größerAls*, da jede dieser *Properties* wiederum auf eine größere verweist. So folgt z.B. aus 3>2 und 2>1 auch 3>1.

Besonders für große Ontologien ist es wichtig, nicht einfach nur *Klassen* zu haben, sondern auch festlegen zu können, was sie definiert. Hierüber lässt sich dann feststellen, in wie weit *Klassen* miteinander in Beziehung stehen. Je größer die Ontologie desto weniger intuitiv ersichtlich kann dieses sein. Häufig ergeben sich bei der Definition durch unbedachte Nebeneffekte allerdings auch Beziehungen, die unerwünscht sind. Dies formalisiert *OWL* durch die Benutzung von Beschreibungslogik für die Klassendefinitionen. Hierdurch wird die Aussagekraft enorm erweitert und es lassen sich maschinengestützt neue Informationen aus einer Ontologie ableiten.

Idealerweise wäre *OWL* eine direkte Erweiterung von *RDFS*, würde also alle bestehenden Elemente nutzen und sie um eigene Komponenten erweitern, um die größere Aussagekraft zu erreichen. Das Problem hierbei ist, dass einige Konstrukte von *RDFS* wie *rdfs:Class* oder *rdf:Property* derartig flexibel sind, dass die Sprache nicht mehr entscheidbar ist. (vergl. [Antoniou & Harmelen 04] S. 113)

Das Problem der Entscheidbarkeit führt dazu, dass es nicht mehr praktikabel möglich ist, *Reasoning* zu betreiben.

Um dieses Problem zu umgehen, wird OWL in drei verschiedene Untersprachen aufgeteilt.

3.1.1 OWL Full

OWL Full wird die vollständige Sprache genannt. Sie ist vollständig kompatibel mit *RDF* und *RDFS*. Das heißt insbesondere, dass es möglich ist, die Bedeutung von vordefinierten Elementen durch die Sprache selbst zu ändern, wie es auch in *RDF* möglich ist. So kann man eine *Restriktion* bezüglich der *Kardinalität* auf die *Klasse*, die alle Klassen beschreibt, anwenden. Das Ergebnis ist, dass hiermit die Anzahl der möglichen *Klassen* limitiert wurde.

Vorteil:
Vollständig aufwärts kompatibel zu *RDF*. Jedes gültige *RDF* Dokument ist auch ein gültiges *OWL Full* Dokument.

Nachteil:
Die Sprache wird durch ihre Möglichkeiten so mächtig, dass sie nicht entscheidbar ist. Dadurch ist es nicht mehr möglich, eine vollständigen bzw. effiziente Unterstützung für das *Reasoning* zu haben.

3.1.2 OWL DL

OWL DL (DL steht für Description Logic, also Beschreibungslogik) ist eine Untersprache von *OWL Full*. Im Wesentlichen beschränkt sie, wie Konstruktoren von *OWL* bzw. *RDF* benutzt werden dürfen. Es ist also nicht mehr möglich, Konstruktoren aufeinander anzuwenden. Das heißt insbesondere, dass es nicht mehr wie bei *OWL Full* möglich wäre, die Gesamtanzahl der Klassen zu limitieren.

Vorteil:
Es ist wieder möglich, wohlbekannte Algorithmen für das *Reasoning* zu benutzen.

Nachteil:
Die allgemeine Aufwärtskompatibilität geht verloren. Zwar ist jedes gültige *OWL DL* Dokument noch ein gültiges *RDF* Dokument, umgekehrt ist das wegen der genannten Beschränkungen aber im Allgemeinen nicht mehr gegeben.

3.1.3 OWL Lite

Abschließend wurde *OWL Lite* spezifiziert. Die Beweggründe hierfür sind diesmal nicht so grundlegender Art wie die Entscheidbarkeitsproblematik bei *OWL DL*. Vielmehr wurde die Sprache hier absichtlich vereinfacht, um sie weniger umfangreich und damit leichter implementiebar zu machen.

Die Unterschiede sind im Einzelnen:

- *Klassen* können nicht als zueinander disjunkt markiert werden
- *Klassen* können nicht aus einer definierten Aufzählung von Instanzen definiert werden (eine sogenannte *enumeratedClass)*
- *Klassen* und *Ranges* von *Properties* können nicht mehr wie bei *OWL DL* durch *descriptions* deklariert werden, welche boolsche Operationen erlauben. So ist es z.B. nicht mehr möglich, die *Range* einer *Property* aus einer Vereinigung mehrerer Klassen aufzubauen oder nur einige, genau definierte *Instanzen* als Möglichkeiten zuzulassen.
- Es werden nur die *Kardinalitäten* 0 und 1 unterstützt.

Vorteil:
Wesentlich einfacher zu implementieren aber trotzdem noch sehr mächtig

Nachteil
Die Ausdrucksmöglichkeiten sind zugunsten des einfacheren Aufbaus beschränkt.

Diese Beschränkung der Ausdrucksstärke muss allerdings nicht unbedingt als Nachteil gesehen werden. Die meisten Anwendungen erfordern viele der mächtigen aussagenlogischen Formulierungen nicht, auch lassen sich komplizierte Sachverhalte häufig recht einfach strukturiert ausdrücken (vergl. [Antoniou & Harmelen 04] S.224).

Aufgrund dessen beschränkt sich die in dieser Arbeit vorgestellte *Ontology Engine* (Kapitel 6) im wesentlichen auf *OWL Lite*. Die vollständige formale abstrakte Spezifikation von *OWL* ist im Anhang, Kapitel A2 zur Referenz enthalten.

Bis auf die genannten Einschränkungen basiert *OWL* direkt auf *RDF* und *RDFS*. So benutzen alle Ausdrücke von *OWL* die Syntax von *RDF*. Auch werden Instanzen genauso wie in *RDF* deklariert. *Klassen* und *Properties* sind eine direkte Erweiterung von denen in *RDF*, wie folgende Abbildung verdeutlicht:

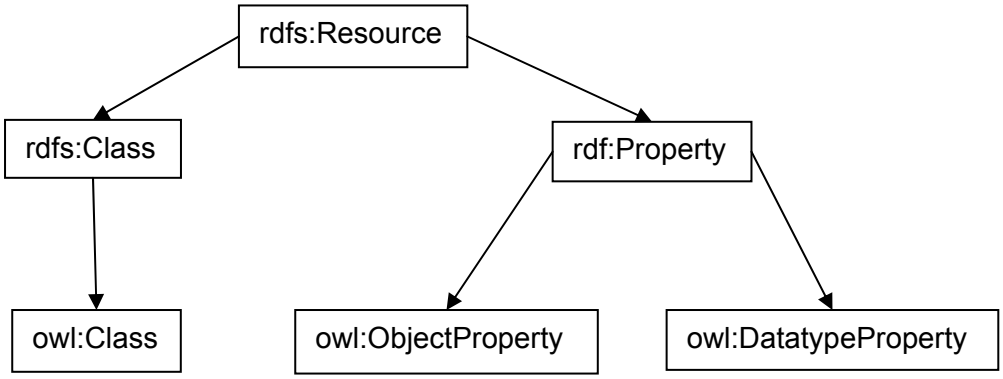


Abbildung 6: Subklassenbeziehung zwischen OWL und RDFS

Die bekannten *rdf:Properties* werden durch *OWL* spezialisiert. Diejenigen, die auf Literale, also Beispielsweise Zahlen oder Zeichenfolgen verweisen, sind in *OWL* die *DatatypeProperties.* Andere, die auf *Instanzen* verweisen, sind in *OWL* die *ObjectProperies.*

Als Beispiel wird die erwähnte *Klasse* **RecreativeUnit** in der textuellen Repräsentation gezeigt:

```
01:  <owl:Class rdf:ID="RecreativeUnit">
02:    <rdfs:subClassOf>
03:      <owl:Restriction>
04:        <owl:allValuesFrom>
05:          <owl:Class rdf:ID="Freizeit"/>
06:        </owl:allValuesFrom>
07:        <owl:onProperty>
08:          <owl:ObjectProperty rdf:ID="has"/>
09:        </owl:onProperty>
10:      </owl:Restriction>
11:    </rdfs:subClassOf>
12:    <rdfs:subClassOf>
13:      <owl:Class rdf:ID="AutonomicUnit"/>
14:    </rdfs:subClassOf>
15:  </owl:Class>
```

Listing 7: RecreativeUnit in OWL/XML

Die *Tags* von *OWL* beginnen mit *owl* und nicht mehr mit *rdf*, die Syntax ist wie erwähnt die gleiche. Die *Restriktion*, dass die **Component** der *Property* **has** nur Elemente der *Klasse* **Freizeit** haben darf, ist etwas trickreich implementiert. Hierzu wird **RecreativeUnit** als *Subklasse* einer so genannten *anonymen Klasse* deklariert, die sich durch eine *Restriction* definiert**.** Diese in den Zeilen 3-10 definierte *Restriction* bedeutet, dass alle Werte der *Property* **has** von der *Klasse* **Freizeit** stammen müssen – unsere gewünschte Einschränkung.

3.2 Bewertung

OWL erweitert *RDFS* um einige sehr mächtige Fähigkeiten und beschränkt *RDFS* andererseits (ab *OWL DL*) so, dass es entscheidbar wird. Weiterhin umgeht *OWL* einige syntaktische und semantische Probleme von *RDFS*, siehe hierzu [Horrocks et al. 03a].

In *RDF* ist es problematisch, dass alle Aussagen in *Triples* aufgebrochen werden, wie auch im Beispiel Kapitel 2.2.3 ersichtlich. Dies ist gerade deshalb problematisch, da diese *Triples* voneinander unabhängig sind, d.h. der Zusammenhang der Aussage, die in mehrere *Triples* aufgebrochen werden musste, geht verloren. Auch kann immer auf alle *Triples* zugegriffen werden, so dass es z.B. möglich ist, eine Aussage die im Kreis läuft zu konstruieren. So kann man eine *Restriktion* auf eine *Property* ausführen, so dass sie nur noch sich selbst als *Range* akzeptiert:

(“Fernseher“, http://organicroom.org/has, “Fernseher“)

Listing 8: Beispiel eines zirkulären RDF Triples

Ein grundsätzliches Problem bleibt, dass *RDF/XML* eine sehr ausführliche Syntax hat. Die im Beispiel definierte *Klasse* **RecreativeUnit** lässt sich mit wenigen Worten beschreiben und auch ihre beschreibungslogische Definition fasst sich recht elegant:

$$RecreativeUnit = AutonomicUnit \sqcap \forall\ has.Freizeit$$

Listing 9: RecreativeUnit in Beschreibungslogik

RecreativeUnits sind also diejenigen **AutonomicUnit**s, deren **has**-Property auf eine *Instanz* der *Klasse* **Freizeit** verweist. Daraus werden durch die *RDF* Syntax, wie oben zu sehen ist, 15 z.T. nicht ganz triviale Zeilen.

Im Hinblick auf die beschränkten Ressourcen der eingebetteten Systeme im *Organic Room* ist dies nicht unproblematisch, zumal für die Verarbeitung derart komplexer Konstrukte auch komplexe Software und viel Rechenleistung und Speicher erforderlich ist.

Insgesamt stellt *OWL* jedoch sowohl aufgrund seiner erweiterten Möglichkeiten aber gerade auch wegen seiner eingeführten Beschränkungen im Vergleich zu *RDFS* einen wichtigen Schritt dar, ontologische Modelle widerspruchsfrei, entscheidbar und maschinenverarbeitbar darzustellen.

3.3 Zusammenfassung

In diesem Kapitel wurde im Detail auf die Anforderungen an eine mächtige Ontologiesprache, wie *OWL* sie darstellt, eingegangen. Es wurden die drei Untersprachen von *OWL* mit ihren Fähigkeiten und Einschränkungen erläutert. Abschließend wurde eine Bewertung im Hinblick auf die Eignung für das Anwendungsgebiet *Organic Room* durchgeführt.

3.4 Literaturempfehlungen

Eine ausführliche Einführung und Überblick über das **Semantische Web** bietet [Antoniou & Harmelen 04]. Vom Mitenwickler von *OWL* ist [Horrocks et al. 03a], welches eine sehr informative Übersicht über die Entstehungsgeschichte bietet. Abschließend bieten die Bachelorarbeit [Abel 04] eine schöne Übersicht, sowie [Noy & McGuinness 01] eine allgemeine Einführung in die Erstellung von Ontologien und was dabei zu beachten ist.

4. Tools für Ontologien

Dieses Kapitel führt kurz in das Tool **Protégé** und sein **OWL Plugin** ein. **Protégé** ist ein Ontologieeditor der Universität Stanford. Er unterstützt den Export nach *OWL RDF-XML* und wird in dieser Arbeit für die Entwicklung der Ontologie des *Organic Rooms* genutzt.

Protégé besitzt ein **OWL Plugin**, das auf **Jena** aufsetzt. Es handelt sich hierbei um ein *Java Framework* für *Semantic Web* Anwendungen. Es bietet eine Programmierschnittstelle für *RDF*, *RDFS* sowie *OWL* und enthält eine regelbasierte Inferenzengine zur Ableitung neuen Wissens anhand der in der Ontologie spezifizierten Regeln.

Dieses Kapitel setzt grundlegende Kenntnis von *OWL* voraus, siehe hierfür Kapitel 2 und 3.

Das Kapitel endet mit einer Bewertung des **OWL Plugins** im Hinblick auf den Einsatz im *Organic Room* sowie einer Zusammenfassung und Literaturempfehlungen.

4.1 Protégé

Im Folgenden wird kurz in die Benutzung von **Protégé**[8] eingeführt. Ziel ist es, die grundlegenden Bedienkonzepte anhand der weiter unten im Detail vorgeführten *Organic Room* Ontologie verständlich zu machen.

Im Folgenden sei darauf verwiesen, dass in **Protégé** die Ausdrücke *Instanz* und *Individual* synonym verwendet werden.

Nach dem Start von **Protégé** erscheint ein Dialog, in dem man vorhandene Projekte öffnen, bzw. neue anlegen kann. Nach dem Öffnen startet **Protégé** mit dem Klasseneditor.

4.1.1 Der OWL Klasseneditor

Zuerst öffnet sich der *OWLClasses* Editor. Hier werden *Klassen* und *Subklassen* erstellt und mit *Properties* versehen.

Im Bild ist die *Klasse* **AutonomicUnit** zu sehen. Sie beschreibt alle Geräte im Raum mit Hilfe ihrer *Properties*.

[8] Protégé ist unter http://protege.stanford.edu/ zu finden.

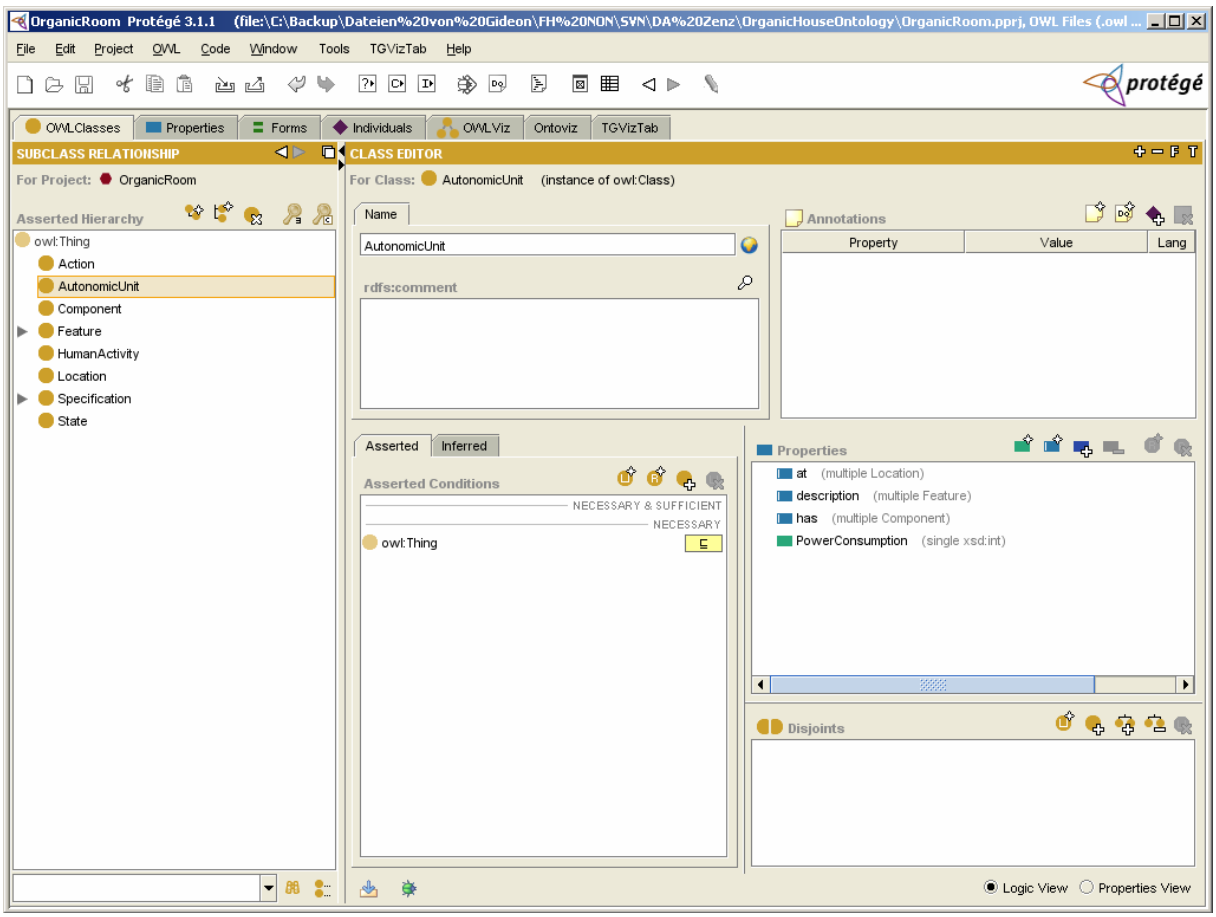


Abbildung 7: Protégé Classes Editor

In *OWL* gibt es grundsätzlich zwei Arten von *Properties*: *Object Properties* (blau in **Protégé**) und *Datatype Properties* (grün in **Protégé**).

Im Beispiel ist *PowerConsumption* eine *Datatype-Property* vom Typ *single* **xsd:int**. Das bedeutet, dass hier ein einzelner Wert vom Typ *Integer* gespeichert wird, in diesem Falle der Stromverbrauch des betreffenden Geräts.

Object-Properties verbinden Instanzen unterschiedlicher Klassen miteinander. So ist z.B. die *Property* **has** vom Typ *multiple* **Component**, d.h. hier können Verweise auf mehrere *Instanzen* der *Klasse* **Component** gespeichert werden.

In *OWL-DL* können *Restriktionen* für die *Properties* definiert werden. Mit Druck auf den Knopf „R+“ kann ein bool'scher Ausdruck definiert werden, der die gewünschte Beschränkung ausdrückt.

4.1.2 Der Property Editor

Hier werden alle definierten *Properties* übersichtlich aufgeführt. Insbesondere wird die Definition nicht mehr wie im **Klasseneditor** in Kurzform dargestellt, sondern ist detailliert zu betrachten.

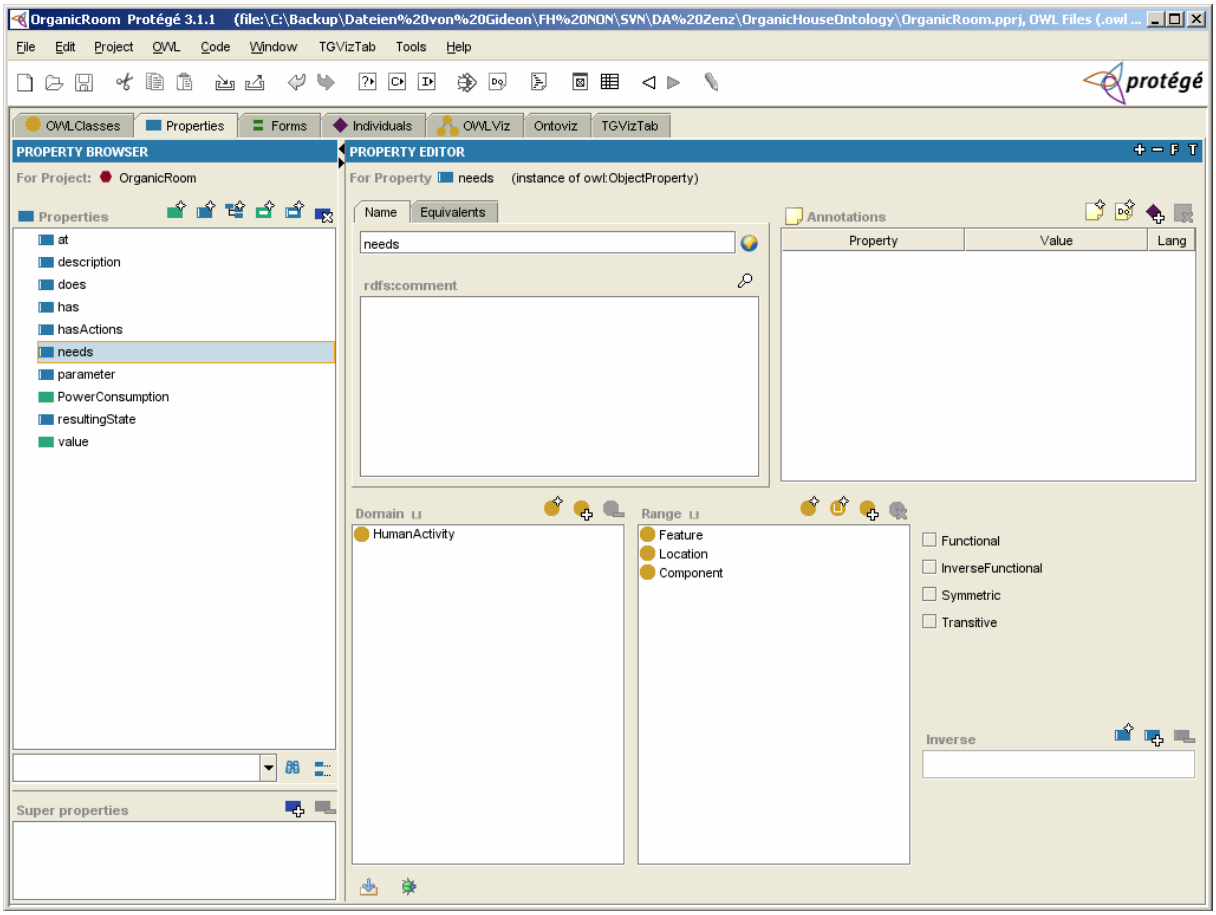


Abbildung 8: Protégé Properties Editor

Das Beispiel zeigt die komplizierteste dieser Ontologie *Property*, nämlich **needs**. Die *Domain* von **needs** ist die *Klasse* **HumanActivity**, d.h. diese Klasse besitzt **needs** als *Property*. Da es sich hier um eine *Object Property* handelt, definiert die *Range*, auf *Instanzen* welcher *Klassen* diese *Property* verweisen kann. Im Beispiel sind hier drei *Klassen* enthalten, und wie das Konjunktionssymbol neben *Range* andeutet, heißt dies, dass *Instanzen* aller drei aufgeführten *Klassen* referenziert werden können.

Weiterhin lassen sich bestimmte **Attribute** festlegen, nämlich **Functional/InverseFunctional** etc. sowie eine oder mehrere *Properties* auswählen, die das **Inverse** der *Property* darstellen sollen (vergl. Kapitel 3).

4.1.3 Der Forms-Editor

In diesem Editor können die Felder im *Individuals*-Editor graphisch angeordnet werden. Er hat also rein praktischen Zweck und modelliert nicht die Ontologie. Aus diesem Grund wird im Rahmen dieser Arbeit nicht weiter auf ihn eingegangen.

4.1.4 Der Individuals-Editor

Im *Individualseditor* werden die Instanzen der vorher definierten *Klassen* bearbeitet bzw. erstellt.

Jede *Instanz verfügt über a*lle *Properties* der Klasse. Es müssen aber natürlich nicht alle benutzt werden.

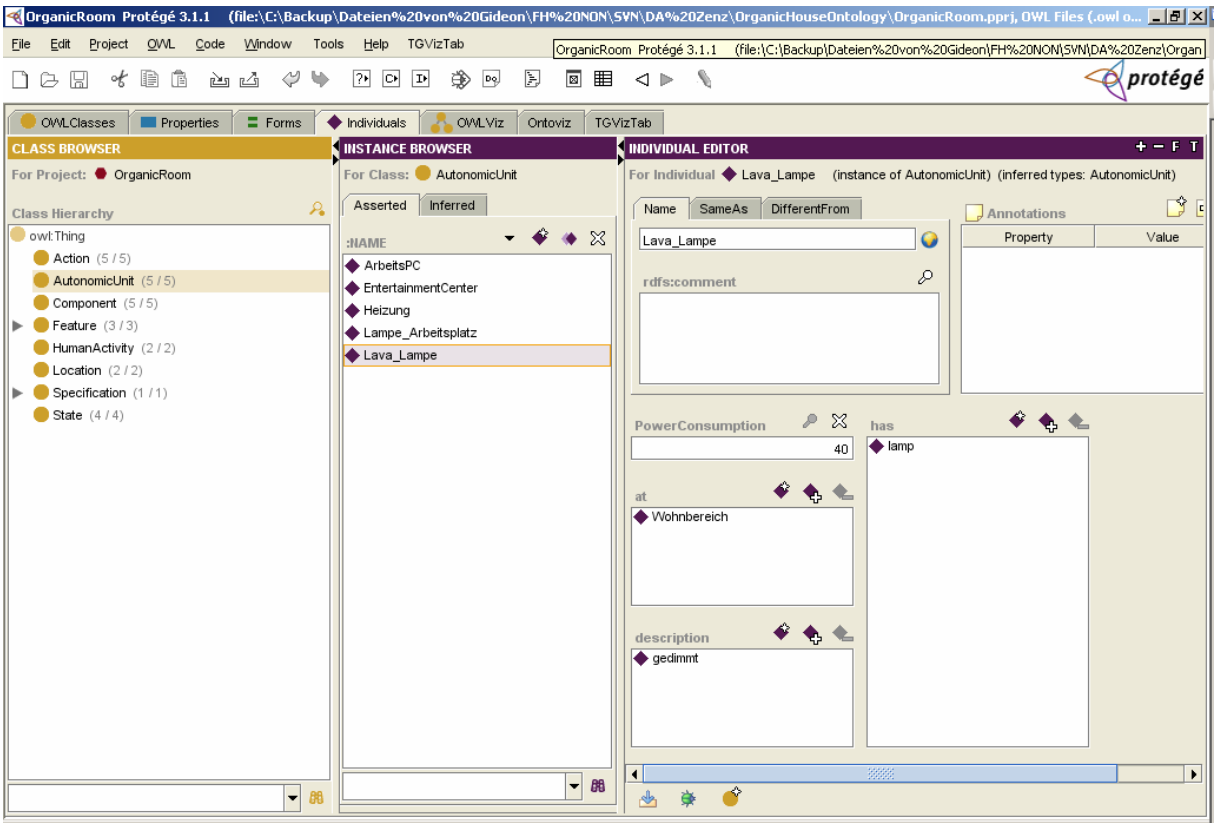


Abbildung 9: Protégé Individuals Editor

Das Beispiel zeigt die *Instanz* **Lava_Lampe** der *Klasse* **AutonomicUnit**. Wie die *Properties* beschreiben, verbraucht sie 40W, steht im **Wohnbereich**, beschreibt sich als **gedimmt** und besitzt eine **Lampe**.

4.2 Das Protege OWL Plugin

Protégé hat ein *Plugin*, das es ermöglicht, *OWL Ontologien* zu erstellen. Es selbst benutzt ein weiteres Paket, **Jena**, innerhalb seines *OWL Plugins* zum Einlesen und Ausgeben von *OWL* Dateien. Dieses Plugin ist insoweit interessant, als es auch aus eigenen Applikationen für diese Zwecke genutzt werden kann. Im Folgenden benutzt der **Importer/Exporter** der **Embedded Ontology Engine** dieses Plugin, um Ontologien von **Protégé** einzulesen bzw. diese wieder mit **Protégé** bearbeitbar zu machen (siehe Kapitel 4.2).

Die Integration von **Jena** in **Protégé** zeigt die folgende Skizze:

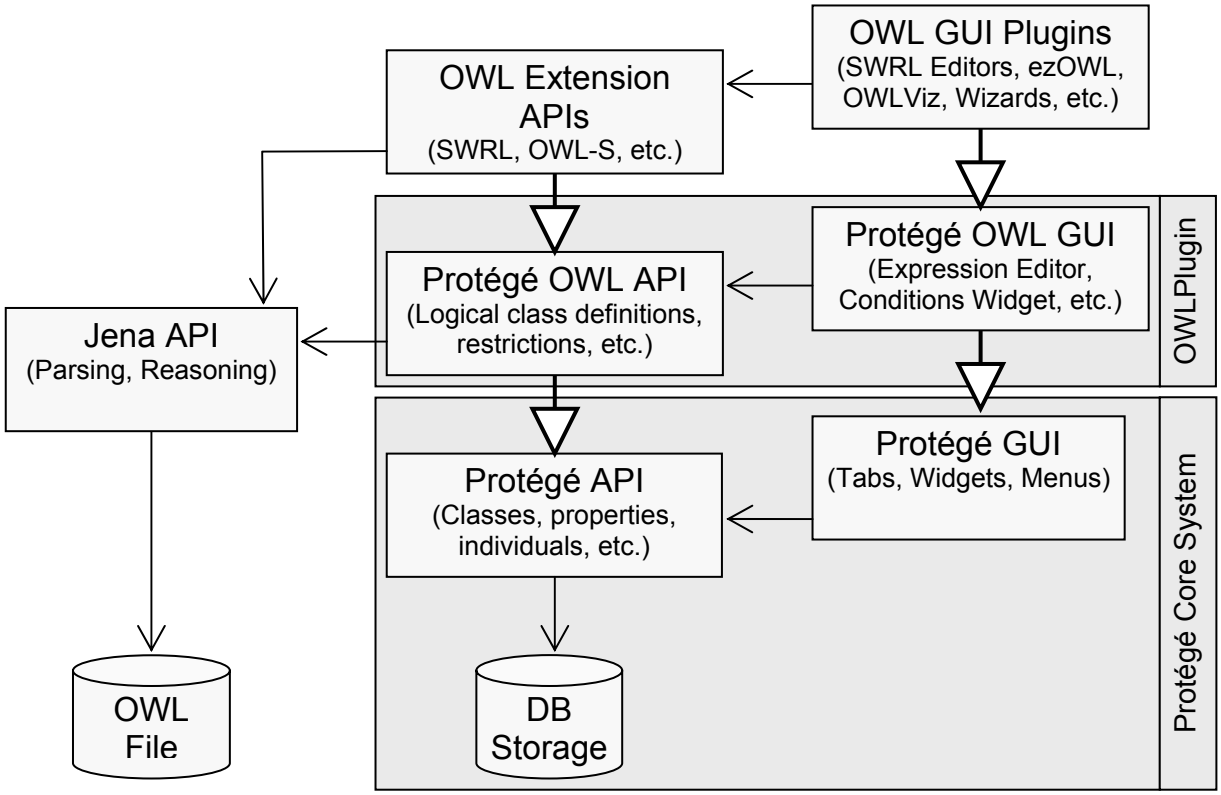


Abbildung 10: Struktur des Protégé OWL Plugins
Aus [Knublauch 04]

Protégé erweitert die **Jena API** zur sogenannten **Protégé OWL API**, was eine konsistente Integration ermöglicht, da **Protégé** teilweise eine andere Nomenklatur als **Jena** verwendet. Dies hat den Vorteil, dass ganz analog zur Benutzeroberfläche auch mittels Software mit der Ontologie gearbeitet werden kann. Als Beispiel dient wieder die Ontologie aus Kapitel 2.3. Es wird exemplarisch die *Klasse* **AutonomicUnit** mit der *Instanz* **Radio** erzeugt.

```
01: OWLModel owlModel = ProtegeOWL.createJenaOWLModel();
02:
03: OWLNamedClass AutonomicUnit =
04:         owlModel.createOWLNamedClass("AutonomicUnit");
05:
06: OWLDatatypeProperty PowerConsumption =
07:         owlModel.createOWLDatatypeProperty("PowerConsumption");
08: PowerConsumption.setRange(owlModel.getXSDint());
09: PowerConsumption.setDomain(AutonomicUnit);
10:
11: OWLObjectProperty has = owlModel.createOWLObjectProperty("has");
12: has.setRange(owlModel.getOWLNamedClass("Component"));
13: has.setDomain(AutonomicUnit);
14:
15: OWLIndividual Radio = AutonomicUnit.createOWLIndividual("Radio");
16: Radio.setPropertyValue(has, owlModel.getOWLIndividual("Musik"));
```

Listing 10: Erzeugen der Klasse AutonomicUnit mit dem Protégé-OWL Plugin

In Zeile 1 wird eine neue Ontologie angelegt. Daraufhin kann in Zeile 3 die erste *OWL Klasse* erzeugt werden. In den Zeilen 6-9 wird die *Property* **PowerConsumption** angelegt. Die *Range* ist **Integer** und über *setDomain()* wird die *Property* der *Klasse* **AutonomicUnit** zugeordnet. **PowerConsumption** ist eine *DatatypeProperty*, da sie sich auf einen Datentyp, nämlich **Integer**, bezieht.

In den Zeilen 11-13 wird die *Property* **has** angelegt. Sie verbindet ein **AutonomicUnit** mit einer oder mehreren *Instanzen* der *Klasse* **Component**. Entsprechend ist sie eine *ObjectProperty*. In diesem Beispiel gehen wir an dieser Stelle davon aus, dass die *Klasse* **Component** bereits erzeugt wurde, was ermöglicht zu zeigen, wie man die Ontologie nach vorhandenen Objekten durchsuchen kann. Dies erledigt die Methode *getOWLNamedClass().* Per *setDomain()* wird die *Property* der *Klasse* **AutonomicUnit** hinzugefügt.

Abschließend wird in der Zeile 15 die *Instanz* **Radio** erzeugt. Da ein Radio Musik erzeugt, wird das **Radio** in Zeile 16 mittels der *Property* **has** mit der *Instanz* **Musik** der *Klasse* **Component** verbunden. Wiederum wird davon ausgegangen, dass **Musik** bereits angelegt wurde. Per *getOWLIndividual()* wird sie aus der Ontologie gesucht.

4.3 Bewertung hinsichtlich verteilter Kleinstsysteme

Protégé stellt ein hervorragendes Tool zur Erstellung und Überprüfung von Ontologien dar. Das **Protégé OWL Plugin** ermöglicht einen komfortablen und umfassenden softwaregestützten Umgang mit Ontologien weshalb es wünschenswert wäre, es auch für den *Organic Room* zu nutzen.

Für das Einsatzgebiet *Embbedded System* bzw. *PDA* muss die verwendete Software jedoch sehr engen Beschränkungen bezüglich des Ressourcenverbrauchs genügen. Weiterhin muss sie auf einer für PDAs verfügbaren *Java Virtual Machine* lauffähig sein. Außerdem muss sich das System für verteilte Ontologien eignen. Das bedeutet, Ontologien schrittweise von anderen Systemen aus dem Netz zu importieren und bei nicht bekannten Teilstücken entsprechend nachzufragen. Insbesondere hierfür bietet das **Protégé OWL Plugin** keine direkte Unterstützung.

Die folgende Übersicht stellt den Ressourcenverbrauch von dem **Protégé OWL Plugin** der in Kapitel 6.3 vorgestellten **Embedded Ontology Engine** gegenüber.

	Protégé OWL Plugin	**Embedded Ontology Engine**	**Faktor**
Zeit für Import der Ontologie	7,3 s	0,13s	50x
Speicherverbrauch	15,4 MiB	1 MiB	15x
Größe der Programmdateien	10,5 MiB	30,9 KiB	350x
Enumerieren aller *Klassen, Instanzen, Properties*	15 ms	0,6 ms	20x

Abbildung 11: Ressourcenverbrauch Protégé OWL - Embedded Ontology Engine

Die Speichermessung wurde über die Methoden der **RunTime** nach vollständigem Einlesen und vorheriger expliziter *Garbagecollection* ausgeführt. Die Messungen wurden auf einem PC mit 2 GiB Speicher und einem **Athlon** XP 2600+ und **SUNs JRE** 1.5.0_06 durchgeführt.

Das **Protégé OWL Plugin** ist mit Java 1.3 entwickelt und somit prinzipiell auf der für den PDA verwendeten *JVM* (siehe Kapitel 6.2.1) lauffähig, was sich nach größeren Modifikationen auch wirklich realisieren lässt. Da der PDA selbst deutlich langsamer als der verwendete PC ist, ist leicht ersichtlich, dass der Einsatz jedoch prinzipiell eher ungünstig ist.

4.4 Zusammenfassung

In diesem Kapitel wurde kurz in das Tool **Protégé** zur Erstellung von *OWL Ontologien* sowie in das auch mit externen Programmen nutzbare **OWL Plugin** eingeführt. Die abschließende Bewertung zeigte den Sinn einer Eigenentwicklung zur Verarbeitung von Ontologien in dem verteilten, dynamischen Szenario des *Organic Room.*

4.5 Literaturempfehlung

Das Tutorial vom Entwickler von **Protégé**, [Knublauch et al. 04], ist eine nicht zu unterschätzende, detaillierte Hilfe bei der Arbeit.
Die API Dokumentation des **OWL Plugins** ist unter [Knublauch 05] zu finden.

5. Modellbildung: Die Organic Room Ontologie

Dieses Kapitel erläutert die Anforderungen an ein intelligentes Haus und leitet hieraus das ontologische Modell des *Organic Rooms* ab. Daraufhin wird dieses Model im Detail durchgegangen.

Das Kapitel endet mit einer Zusammenfassung sowie Literaturempfehlungen.

5.1 Einführung

Der Mensch strebt seit jeher danach, sein alltägliches Leben möglichst angenehm und komfortabel zu gestalten. Da dieses hauptsächlich in Gebäuden stattfindet, ist Gebäudetechnik dabei ein wichtiger Faktor. Die Optimierungsmöglichkeiten einzelner Geräte sind inzwischen auch sehr weitgehend ausgeschöpft, die Möglichkeiten der Vernetzung dagegen noch kaum, und das, obwohl erste Ideen zur vernetzten Haustechnik schon aus den 60ern stammen (Vergl. Projekt *Xanadu*, [Broy et. Al 00]).

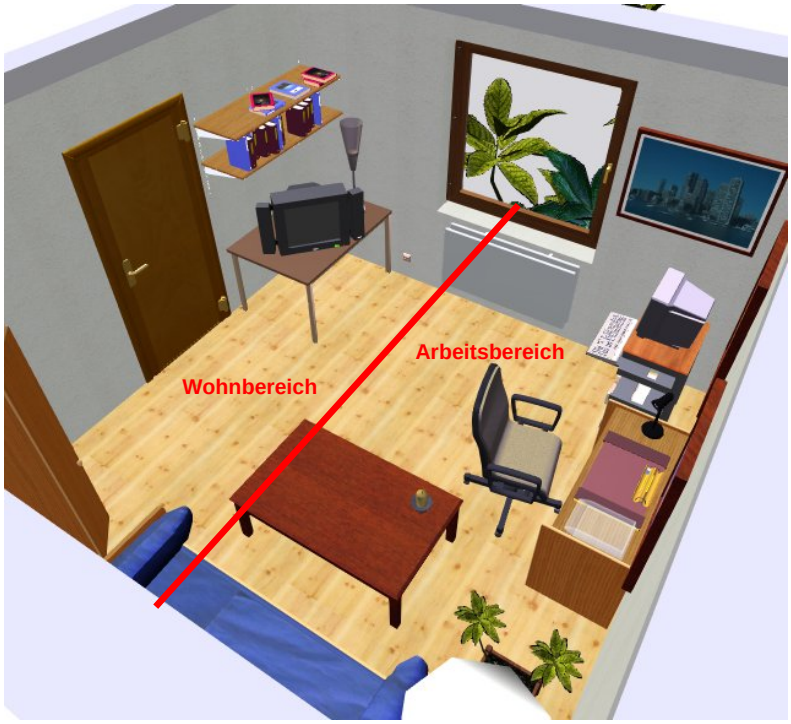


Abbildung 12: Der Organic Room

Diese ermöglichen, Szenarien zu definiert, wie z.B. „Haus verlassen“, auf das alle Geräte des Hauses entsprechend reagieren. Sie sollen aber nicht starre Aktionen ausführen, sondern konfigurierbar sein. Entsprechende Technologien werden z.B. im *inHaus* des Fraunhoferinstitutes für Mikroelektronische Schaltungen und Systeme getestet. (vergl. [Scherer & Grinewitschus 02]). Eine zentrale Rolle spielt, dass Geräte durch ein zentrales mobiles Gerät, wie z.B. einen *PDA* gesteuert werden und nicht mehr über einzelne Panels oder Fernbedienungen, was die Bedienung erheblich erleichtert.

Der Ansatz des *Organic Room* nimmt diese Gedanken auf und entwickelt sie weiter, insoweit die Geräte ihre Fähigkeiten in assoziativer Weise kommunizieren können. Hierzu ein Beispiel:

> Person A möchte in sein Zimmer gehen und ein Buch lesen. Traditionell würde er oder sie das Zimmer betreten, ein **helles** Licht anschalten, möglichst in der Nähe des **Arbeitsplatzes**. Da der Raum ungenutzt war, ist die Heizung heruntergestellt. Jetzt sollte es aber angenehm **warm** sein, also wird sie entsprechend eingestellt. Je nach Geschmack ist evtl. auch etwas ruhige **Musik** als Hintergrunduntermalung gewünscht. Jetzt ist der Raum komfortabel „parametriert“ und es kann mit dem lesen begonnen werden.

Abbildung 13: Beispiel Aktivität „lesen“

Der *Organic Room* soll diese Parametrierung möglichst eigenständig aus der gewünschten Aktivität „*lesen*“ nach den Bedürfnissen des Nutzers ableiten können. Im Beispiel wurden die Attribute, die mit der Aktivität verbunden sind, hervorgehoben. Die Aktivität *Lesen* verlangt also nach Geräten, die über die Attribute **hell, Arbeitsplatz, warm** sowie **Musik** verfügen. Solche Geräte soll der *PDA* des Benutzers eigenständig finden und sinnvolle Aktionen vorschlagen können. Dieser Vorgang wird **matching** genannt.

Der Benutzer soll die Aktionen weiterhin bei Bedarf nach seinen Wünschen im Detail konfigurieren können (z.B. welche **Musik** denn genau, oder welchen Stil, nicht nur das Musik gewünscht ist) und die Ausführung veranlassen können. Die gewählte Detailkonfiguration soll für diese Geräte und diesen Raum gespeichert werden.

Wichtig hierbei ist, dass *PDA* und Raum sich vorher nicht kennen müssen. Alle Informationen müssen dynamisch gewonnen und Entscheidungen anhand der Beschreibung der Geräte gefunden werden.

Eckpunkte sind also:

- Der *PDA* des Benutzers muss alle Geräte im Raum finden
- Die Geräte müssen ihre Fähigkeiten dem *PDA* mitteilen
- Der *PDA* muss anhand von Attributen für Tätigkeiten des Benutzers passende Geräte finden und für diese Geräte passende Aktionen

Abbildung 14: Anforderungen an die Organic Room Ontologie

Ontologisch gesehen teilt sich der Raum also in zwei Hauptaspekte auf: Die **Geräte** mit ihren Funktionen und die **Aktivitäten** des Menschen mit seinen Bedürfnissen. Geräte und Aktivitäten definieren sich hierbei über Attribute, die sie beschreiben bzw. benötigen, nämlich der **Ort** an dem sie stehen, die **Fähigkeiten**, über die sie Verfügen (wie **Helligkeit** oder **Wärme** zu erzeugen), sowie die konkreten technischen **Komponenten**, aus denen sie bestehen (wie eine **Lampe** oder eine **Heizeinheit**).

Weiterhin muss aus der Beschreibung der **Komponenten** hervorgehen, wie diese zu bedienen sind und was konkret sie können.

Eine Ontologie, die das Geforderte leistet, ist in Abbildung 15 zu sehen. Sie orientiert sich in ihren Grundsätzen an die in [Mahmoudi & Müller-Schloer 06] vorgeschlagene *Kernelontologie*, greift aber auch die Idee von [Hendler 01] auf, die Beschreibung von Zustandsautomaten mit in die Ontologie zu integrieren.

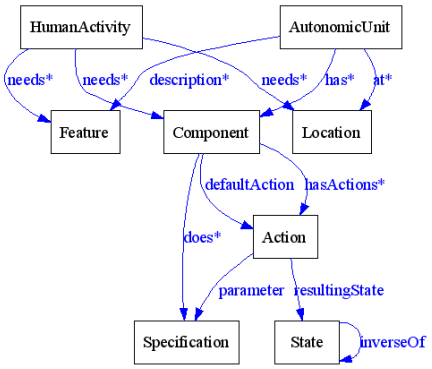


Abbildung 15: Klassendiagramm Organic Room Ontologie

Aktivitäten (*HumanActivity*) benötigen und Geräte (*AutonomicUnit*) haben jeweils Fähigkeiten (**Feature**), Komponenten (**Component**) sowie eine Örtlichkeit (**Location**).

Komponenten beschreiben wiederum einen Zustandsautomaten, der ihre Funktionalität spezifiziert. Sie besitzen Aktionen (**Action**), die zu einem neuen Zustand (**State**) führen. Weiterhin erfüllen sie eine bestimmte technische Spezifikation (**Specification**).

Die *Klassen* und ihre *Instanzen* im Detail:

5.2 Klasse HumanActivity

Diese *Klasse* beschreibt alle möglichen Aktivitäten des Benutzers. Im *Organic Room* sind dies **Filme** sehen, **Musik** hören, **Programmieren**, **Schreiben** sowie **Entspannen**. Eine Aktivität benötigt gewisse Dinge, nämlich Geräte mit bestimmten Eigenschaften an einem bestimmten Ort. So benötigt man zum **Programmieren** z.B. **Entwicklungssoftware** (*Component*) am **Arbeitsplatz** (*Location*) sowie eine **helle** Beleuchtung (Feature).

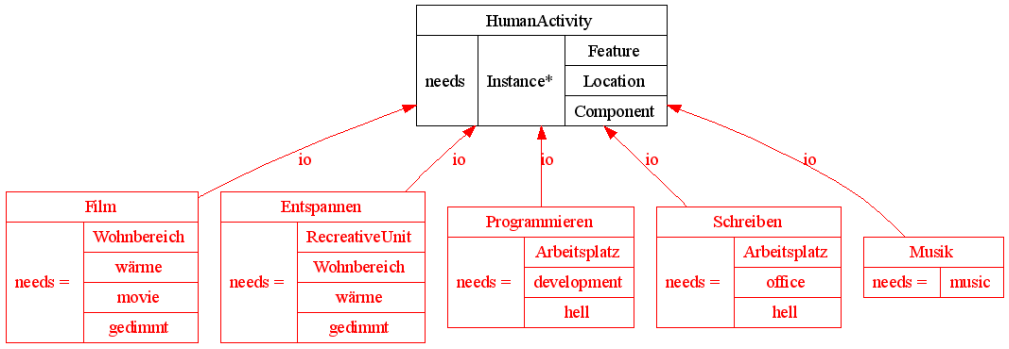


Abbildung 16: Klasse HumanActivity

5.3 Klasse AutonomicUnit

Das Pendant zu **HumanActivity** ist **AutonomicUnit**. Hier werden die Geräte mit ihren Komponenten und Eigenschaften beschrieben. Weiterhin ist ein Gerät einem Ort zugeordnet und hat einen bestimmten Energieverbrauch.

So verbraucht ein **ArbeitsPC** beispielsweise 300W, hat die Komponente **Software**, beschreibt sich mit der Eigenschaft **ProductiveUnit** und steht am Ort **Arbeitsplatz**.

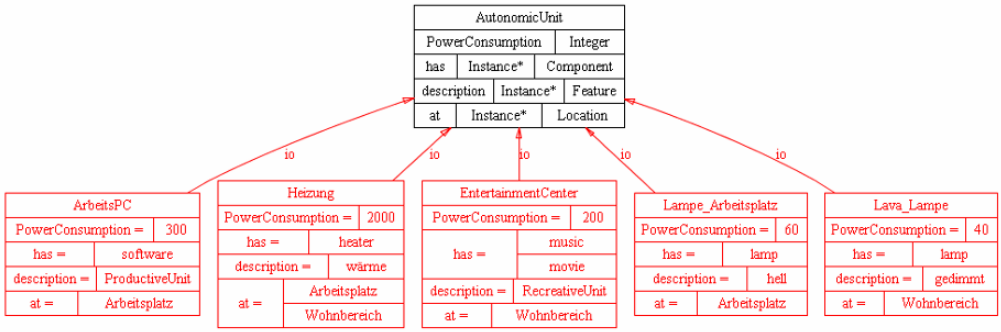


Abbildung 17: Klasse AutonomicUnit

5.4 Klasse Feature

Features sind Eigenschaften von Geräten, im Gegensatz zu **Specifications**, die Eigenschaften von Komponenten beschreiben. So bieten sich z.B. die *Instanzen* **hell** oder **gedimmt** für Lampen an. Die *Subklasse* **ComplexUnit** ist für Geräte mit mehreren Funktionen gedacht, die einem bestimmten Zweck dienen. Hier sind derzeit das **ProductiveUnit** (Arbeitsrechner) sowie das **RecreativeUnit** (Entertainmentcenter) enthalten.

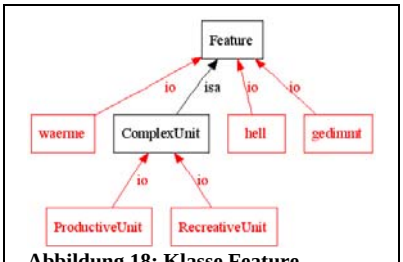


Abbildung 18: Klasse Feature

5.5 Klasse Location

Die *Klasse* **Location** beschreibt, an welchem (logischen) Ort sich ein Gerät befindet. Dies erlaubt z.B. bei gewünschter Tätigkeit **Arbeit** alle zu dem **Arbeitsplatz** gehörenden Geräte dem Benutzer aufzulisten oder beispielsweise nur die Lampen anzuschalten, die an diesem Platz sind.

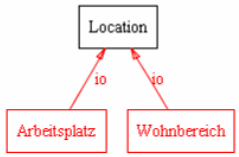


Abbildung 19: Klasse Location

5.6 Klasse Component

Die *Klasse* **Component** beschreibt *Komponenten* von Geräten wie eine **Lampe**, **Software**, **Musik** etc. Alle *Komponenten* erfüllen bestimmte *Spezifikationen*, so erzeugt eine **Lampe Licht**, **Software** stellt Anwendungs-, Büro- oder Grafik**software** zur Verfügung etc.

Weiterhin gibt es *Aktionen*, die man mit einer *Komponente* ausführen kann, wie z.B. das *Schalten* bei der **Lampe** oder *Starten/Stoppen* von **Software** bzw. das *Listen* der verfügbaren **Software**.

Für das *Matching* wird hier auch eine *defaultAction* spezifiziert. Sie gibt vor, mit welcher *Aktion* ein Gerät in Betrieb zu nehmen ist.

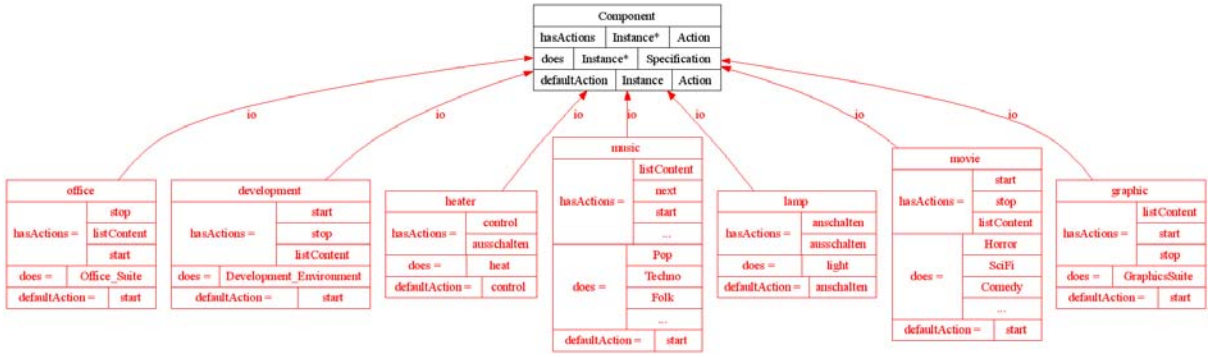


Abbildung 20: Klasse Component

5.7 Klasse Action

Action beschreibt *Aktionen*, die *Komponenten* ausführen können. *Aktionen* können *Parameter* haben, wie z.B. **control**, welches für den **Heizungsregler** gedacht ist, und ein *resultingState*, der alle möglichen Ergebnisse dieser Aktion beinhaltet. Der **Parameter** gibt an, dass dem Befehl ein Parameter übergeben werden muss und spezifiziert den Typ. Bei **control** ist das die Ziel-Temperatur, bei **start** gibt er den Typus des zu startenden Mediums an. Siehe hierzu Kapitel 5.9.

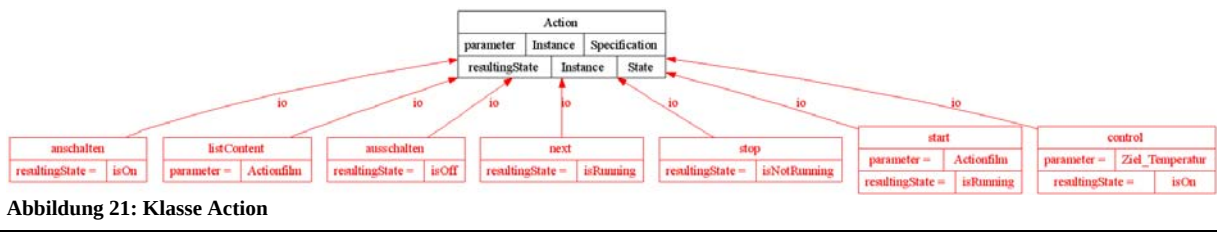


Abbildung 21: Klasse Action

5.8 Klasse State

State beschreibt die möglichen Zustände einer *Komponente*, sowie das *Inverse* dieser Zustände. Damit kann über die assoziierten *Components* eine Gegenaktion für den Zustand einer *Component* gefunden werden, z.B. um sie auszuschalten, wenn sie angeschaltet ist.

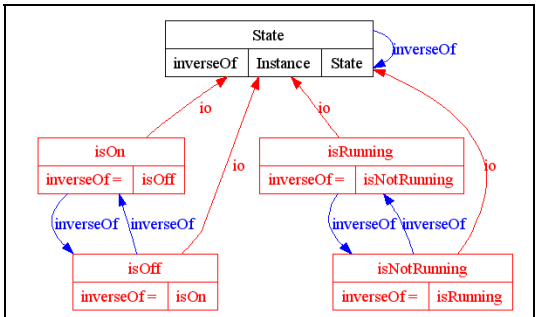


Abbildung 22: Klasse State

5.9 Klasse Specification

Specification beschreibt messbare Werte oder deskriptive Elemente einer *Komponente*. Für den **Heizungsregler** ist das z.B. die **Zieltemperatur**, für **Musik** die Geschwindigkeit oder den Typ der Musik. Dies ermöglicht eine Vorgabe zu machen, anhand der das Gerät ein passendes Stück finden kann[9]. Die Möglichkeiten sind: **Actionfilm, Bluesrock, Comedy, Develompent_Environment, Dokumentarfilm, Fantasy, Folk, GraphicsSuite, HipHop, Horror, Hörbuch, Klassik, Office_Suite, Pop, Romantik, SciFi** sowie **Techno**.

Weiterhin ist es möglich eine **value** anzugeben, die einen Standardwert vorgibt, wie Beispielsweise bei der **Ziel_Temperatur**. Eine Angabe eines solchen Wertes bedeutet weiterhin für die auf diese *Specification* verweisende *Action*, das hier ein entsprechender Wert, also hier eine Zahl, als *Parameter* gefordert ist.

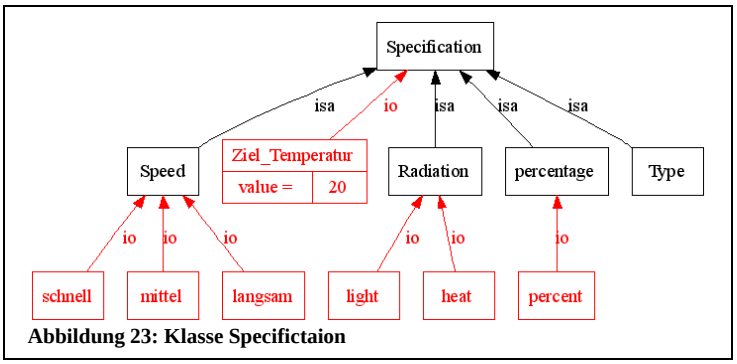


Abbildung 23: Klasse Specifictaion

5.10 Zusammenfassung

Dieses Kapitel hat in die Anforderungen an ein vernetztes Haus, insbesondere des *Organic Rooms* und der zentralen Steuerbarkeit vorhandener Geräte sowie der einheitlichen Reaktion von Geräten auf Tätigkeiten des Benutzer eingeführt.

Weiterhin wurde eine Ontologie vorgestellt, die die genannten Anforderungen erfüllt und als Basis für die in Kapitel 6.5 vorgestellte Software dient.

5.11 Literaturempfehlungen

Eine Hervorragende Übersicht über die Thematik intelligenter Haustechnik bietet [Broy et al. 00]. Die Methodik zur Modellierung von Ontologien wird in [Noy & McGuinness 01] vertieft behandelt.

[9] **Type** ist wegen der Anzahl der *Instanzen* nicht grafisch dargestellt.

6. Implementierung

6.1 Einführung

In Kapitel 1 wurde der *Organic Room* mit seinen Anforderungen eingeführt. In Kapitel 5 wurde daraufhin eine Ontologie entwickelt, die diesen Anforderungen gerecht wird.

Bereits vorgestellt sind:

- Eine Sprache zur Speicherung und Verarbeitung von Wissen (Kap. 3)
- Eine Methodik zur Erzeugung und Abfrage von Wissensdatenbanken (Kap. 4 und 5)

Es fehlt:

- Eine Netzwerktechnik, die dynamisch Geräte im Raum erkennt
- Ein Kommunikationsprotokoll, das das gespeicherte Wissen verteilt
- Eine Methodik, Aktivitäten des Nutzers zu erfragen
- Eine Methodik, intelligent zu den Aktivitäten passende Geräte zu identifizieren, zu parametrieren und in Betrieb zu nehmen
- Passende Geräte hierfür

In diesem Kapitel werden zunächst passende Geräte vorgestellt, nämlich ein *PDA* in Kapitel 6.2.1, das es dem Benutzer per Funk ermöglicht, andere Geräte seiner Umgebung zu finden und ihm über den Bildschirm die Möglichkeit zur Interaktion bietet.

Weiterhin wird ein kleines Gerät benötigt, das Wissen kommunizieren und Verbraucher schalten kann. Es fungiert quasi als *Zwischenstecker,* beispielsweise für eine Lampe oder einen PC. Der *PDA* soll mit ihm in Verbindung treten können und alle notwendigen Informationen über das Gerät, das der *Zwischenstecker* betreibt, erhalten können. Wenn der *PDA* dann das Gerät in Betrieb nehmen will, schaltet der *Zwischenstecker* es ein und parametriert es gegebenenfalls. Kapitel 6.2.2 stellt die hierfür verwendete Plattform vor.

Da der Benutzer sich mit seinem *PDA* ungebunden durch die Räumlichkeiten bewegt, wird ein sehr dynamisches Netzwerkprotokoll benötigt. Es muss erlauben, plötzlich auftauchende Geräte zu identifizieren, es muss darauf vorbereitet sein, dass Geräte ebenso plötzlich wieder verschwinden. Weiterhin muss es konfigurationsfrei arbeiten, damit der Umgang mit dem *Organic Room* ein für den Benutzer befriedigendes Erlebnis bietet. Diese Problematik wird in Kapitel 6.2.3 diskutiert.

Weiterhin wird in Kapitel 6.3 die sogenannte *Embedded Ontology Engine* vorgestellt. Schon in Kapitel 4.3 wurde klar, dass alleine aus Performancegründen eine eigene Software zur Verarbeitung der Ontologie geschaffen werden muss. Die Anforderungen an diese werden weiter präzisiert und eine Lösung vorgestellt.

Schließlich wird noch eine Software benötigt, die die *Human-Machine* Kommunikation übernimmt, sprich dem Benutzer die gefundenen Geräte und Leistungen präsentiert, seine Aktivitäten ermittelt und passend dazu Geräte vorschlägt und diese steuern kann. Dies wird ab Kapitel 6.5 diskutiert.

Das Kapitel endet mit einer Zusammenfassung sowie mit Literaturempfehlungen.

6.2 Verwendete Geräte

6.2.1 PDA als Human-Machine Interface

> „Eine wichtige übergeordnete Anwendung besteht in der Bereitstellung einer integrierten Bedienung für das Haustechnik-System. Separate Bediengeräte inkl. deren unterschiedlichen Bedienlogiken sind in einem Haussystem nicht mehr tolerabel. Wegen der massenhaften Verbreitung, der günstigen Preise und der bekannten Bedienkonzepte wird zukünftig die Multimedia-Technik eine wichtige Rolle bei der integrierten Systembedienung einnehmen. Es wird daran gearbeitet, die Haustechnik über den PC, den Fernseher, über Notepads und auch über PDA´s und Smartphones bedienbar zu machen."
> [Scherer & Grinewitschus 02]

Der *PDA* stellt für uns aus mehreren Gründen eine favorisierte Lösung dar. PDAs sind relativ leicht und komfortabel mit sich zu führen. Weiterhin sind sie schon direkt mit Funktechnik (WLAN / Bluetooth) ausgestattet erhältlich, was die dynamische Vernetzung mit Räumen erst ermöglicht. Außerdem sind sie inzwischen sehr leistungsstark und verfügen über ein großes, per Stift gut bedienbares Display. Weiterhin ist auf ihnen mit dem **IBM WebSphere Everyplace Micro Environment**[10] eine leistungsfähige Java Entwicklungsumgebung vorhanden, die einfache Entwicklung ermöglicht.

Für diese Arbeit wurde ein **Dell Axim X51v** benutzt. Er besitzt eine **Intel Xscale** CPU mit 624 MHz, 64 MB RAM und 256 MB Flashspeicher. Weiterhin ist der Bildschirm mit 480x640 Pixeln hochauflösend.
Als Betriebssystem wird **Microsoft Windows Mobile 5** verwendet.

Nebenstehende Abbildung zeigt das Gerät mit der in Kapitel 6.5 vorgestellten *Organic Room Software.*

Abbildung 24: Der verwendete PDA von DELL

[10] IBM WebSphere Everyplace Micro Environment ist unter http://www-306.ibm.com/software/wireless/weme/ zu finden.

6.2.2 Infineon XC 167 CI als Machine-Machine Interface

Als Plattform für den *Zwischenstecker* kommt der **Infineon XC167CI** zum Einsatz. Es handelt sich hierbei um einen 16 Bit Prozessor mit 40 Mhz, der hochkonfigurabel und ideal zur Ansteuerung von externer Peripherie ist. Auf dem verwendeten Entwicklungsboard findet sich noch ein **CS8900a** Ethernet Baustein sowie 512KiB SRAM und ebenso viel Flashspeicher.

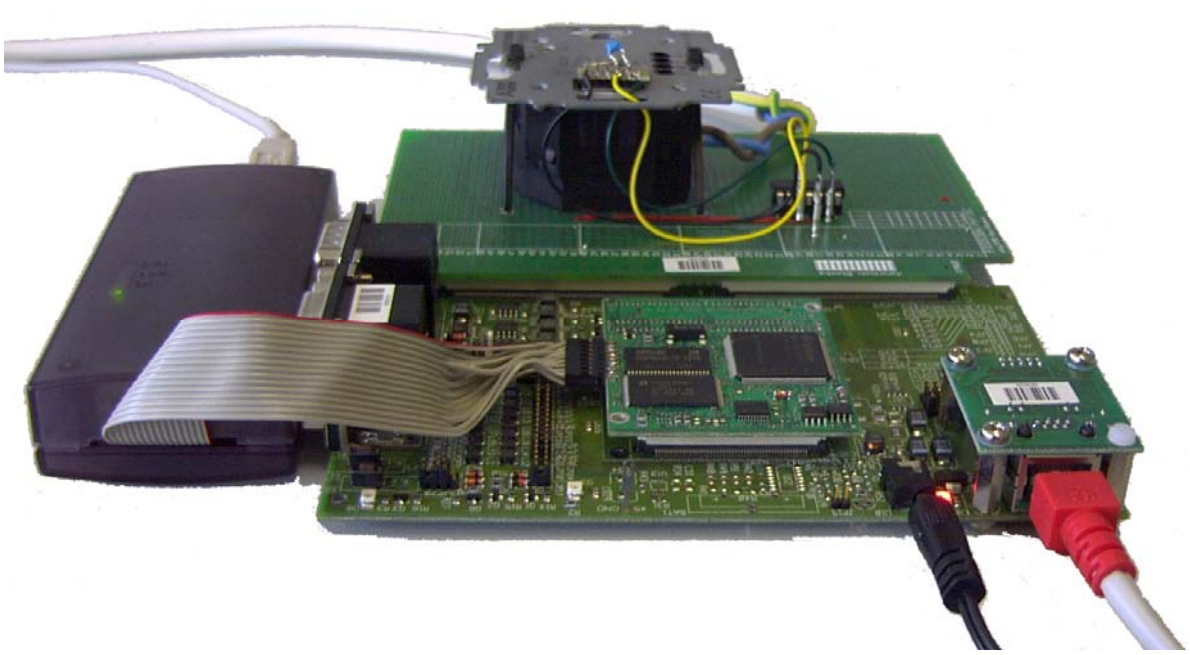


Abbildung 25: Infineon XC167 CI Entwicklungsboard mit JTAG Adapter und Relay

Weiterhin wurde an das Board ein Relay der Firma **Busch-Jäge**r angeschlossen, mit dem Verbraucher schaltbar sind. Die Verbindung zum Funknetz erfolgt zentral über einen Accesspoint.

Als Software kommt der **VISE TCP/IP Stack** zum Einsatz. Es handelt sich hierbei um ein **SDL.RT** basiertes Betriebssystem, auf dem ein modularer TCP/IP Stack aufgesetzt wurde.

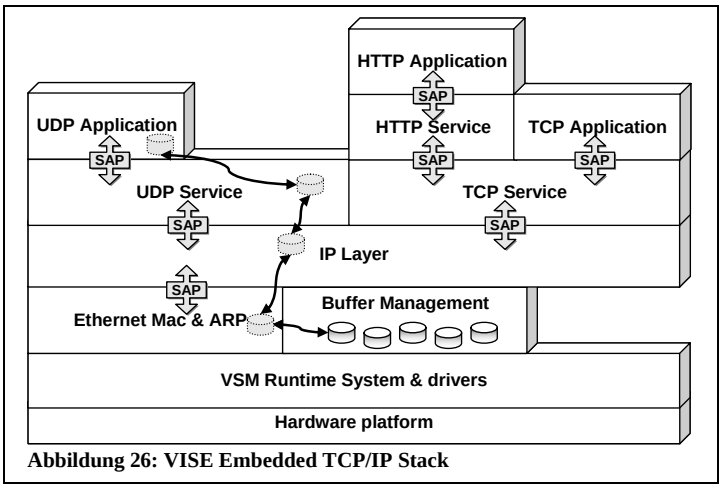


Abbildung 26: VISE Embedded TCP/IP Stack

Aus [Welge 06]

Die Skizze zeigt den internen, modularen Aufbau des Stacks. Jede Ebene kommuniziert mit der nächsten über *Service Access Points*. Hierfür registriert die jeweilige Ebene eine *Callback* Funktion, die im Ereignisfall aufgerufen wird. Der Stack ist dadurch dynamisch erweiterbar.

6.2.3 L3 Ad-Hoc Netzwerk

Wie eingangs erwähnt wurde, ist für das Szenario des *Organic Room* ein Netzwerkprotokoll von Nöten, das hochdynamische Netze unterstützt. Eine Gattung, die diesen Anforderungen Rechnung trägt, sind die *Peer-to-Peer (P2P)* Netzwerke.

Im Gegensatz zum klassischen Client/Server-Modell bei Netzwerken ermöglicht *P2P* jedem netzwerkfähigem Gerät *Services* anderen netzwerkfähigen Geräten zur Verfügung zu stellen. Bisher verbindet sich ein Client mit einem Server mittels Kommunikationsprotokolle wie *FTP* oder *HTTP* um Zugriff auf die gewünschten Daten zu erhalten, wobei die größere Arbeitslast bei den Servern liegt, während die Clients relativ wenig ausgelastet werden.

Diese Architektur hat den großen Nachteil, dass mit steigender Anzahl der Clients die Last auf dem Server und die vom Server benötigte Bandbreite proportional steigen. Dies verlangt immer aufwendigere Hardware auf der Serverseite und immer komplexere Verwaltungsmechanismen.

Hier zeigt *P2P* seinen großen Vorteil. Im Gegensatz zu dem zentralisierten Client/Server Modell wird ein dezentrales Netz aufgebaut, das eine flache Struktur mit hoher Interkonnektivität bietet, da hier jeder Rechner auch als Server fungiert, indem er entweder eigenständig Daten zur Verfügung stellt, sie weiterleitet oder zwischenspeichert.

Dies verteilt sowohl die Rechenlast als auch die benötigte Bandbreite und ermöglicht es so zu insgesamt sichereren, verfügbareren und kostengünstigeren Systemen zu gelangen. Sicherer, da wegen der Verteilung Redundanz entsteht, verfügbarer, da nicht alles an einem einzigen Knoten hängt, sondern verteilt ist, und schließlich kostengünstiger, da nicht mehr hochspezialisierte Hardware gebraucht wird, sondern auch billigste Einzelsysteme gemeinsam ein hochleistungsfähiges Netz aufbauen können.

Weiterhin organisieren sich *P2P* Netzwerke eigenständig. Dies funktioniert, indem jeder Peer (so wird der Client/Server-Zwitter bei *P2P* Netzen genannt) sich in seiner Umgebung umhört, ob noch andere Peers vorhanden sind. Dieser Vorgang nennt sich *Discovery* und ermöglicht, dynamisch auftauchende bzw. verschwindende Peers zu erkennen und ihre *Services* zu identifizieren.

Durch diesen Automatismus spannt sich ein eigenständig organisiertes, hochdynamisches Netz auf. Besonders im drahtlosen Bereich (*Ad-Hoc* Netzwerke) ist dies ein großer Vorteil.

In diesem Bereich servicebasierter *Ad-Hoc* Netzwerke existieren bereits einige Lösungen. Besonders sei hier **JXTA** der Firma **SUN** erwähnt. Es ist sehr breit angelegt und versucht alle Anforderungen an *P2P* Netzwerken zu vereinen.

> „*JXTA* technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless *PDAs* to *PCs* and servers to communicate and collaborate in a *P2P* manner.“ [11]

JXTA setzt auf *XML* als zentrale Komponente und *TCP* bzw. *HTTP* zum Nachrichtentransport. Der breite Ansatz, die verwendeten Techniken sowie die Implementierung bringen jedoch den Nachteil mit sich, das sie sehr ressourcenbedürftig sind, wie [Halepovic & Deters 03] zeigen.

[11] http://www.jxta.org

Aus diesem Grund setzt diese Arbeit auf dem *L3 Netzwerkprotokoll* auf. *L3* steht für *Low Power, Low Cost, Low Datarate* und ist ein Protokoll, das die prinzipiellen Möglichkeiten von *JXTA* beinhaltet, jedoch deutlich schmaler aufgestellt ist. Kernelement ist dabei, das nicht *XML* sondern ein binäres Format für die Datenübertragung verwendet wird und die Implementierung selbst mit Hinblick auf beschränkte Ressourcen hinsichtlich Speicher und Rechenleistung entwickelt wurde.

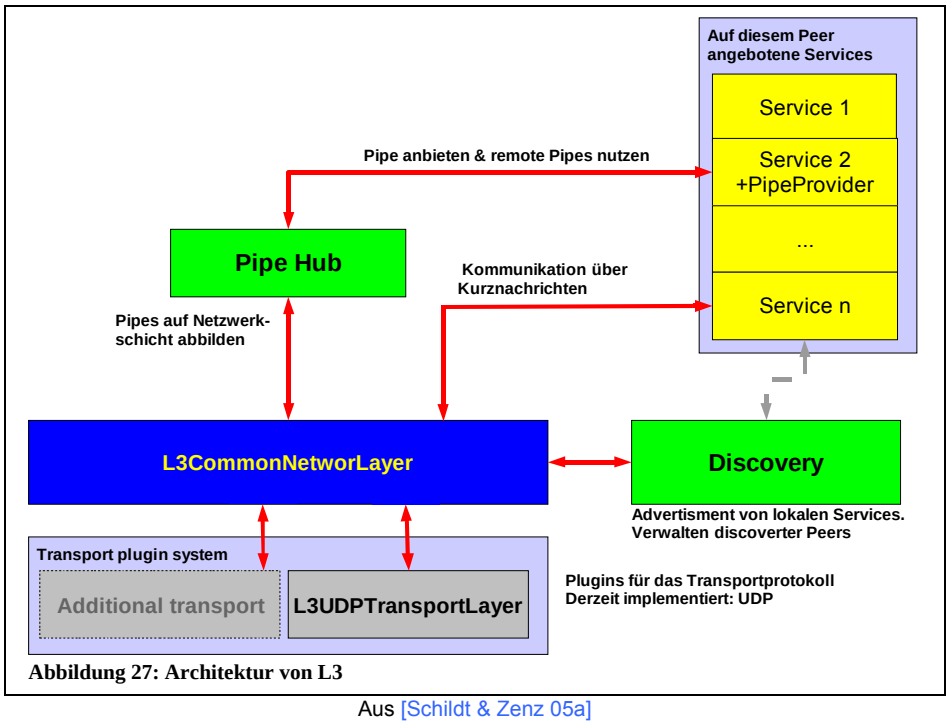


Abbildung 27: Architektur von L3

Aus [Schildt & Zenz 05a]

Aufgrund der Ausrichtung dieses Protokolls existieren Implementierungen sowohl für *Java* als auch für *Embedded C*. Diese Versionen wurden an die vorgestellten Plattformen *J2ME* sowie *VISE* angepasst.

6.3 Die Embedded Ontology Engine

6.3.1 Anforderungen

Wie schon ausführlich diskutiert, bestehen an eine Wissensdatenbank-Engine für den *Organic Room* sehr spezielle Anforderungen. Sie muss folgendes erfüllen:

- Ressourcenschonend
  Die verwendeten Plattformen sind stark ressourcenbeschränkt, insbesondere damit auch die Netzwerkkapazitäten.

- Robust
  Ontologien erweitern sich durch das Finden neuer Geräte, was aber kontrolliert geschehen muss. Die Ontologie darf weder inkonsistent werden noch unkontrolliert wachsen.

- Dynamisches Verteilen von Wissen
  Geräte müssen ihre ontologischen Beschreibungen untereinander austauschen. Idealerweise sollte dies ökonomisch geschehen, d.h. es wird nur das Notwendige übertragen und bei Bedarf nachgefragt.

- Schnittstellen zu Standardtools
  Es sollte möglich sein, eine mit mächtigen Werkzeugen wie **Protégé** erstellte Ontologie zu nutzen. Weiterhin sollte die Ontologie aus dem verteilten System wieder mit **Protégé** analysiert werden können.

Im Folgenden wird auf diese Anforderungen im Detail eingegangen.

6.3.1.1 Ressourcenbedarf

Wie schon in Kapitel 3.2 erwähnt, ist ein Problem von *OWL-XML* die Ausführlichkeit der Repräsentation. Schon kurze *Statements* verwandeln sich in sehr lang gewundene Ausdrücke mit vielen Elementen, was nicht nur den Platzbedarf erhöht, sondern auch das Verarbeiten kompliziert. Ideal wäre deshalb eine Repräsentation von *OWL,* die sich auf das Wesentliche beschränkt.

Weiterhin sollte diese Repräsentation möglichst direkt in eine sinnvolle und performante Speicherrepräsentation überführbar sein. In der **Ausführungsphase** der Ontologie kommt neben möglichst geringem Platzbedarf noch die Anforderung hinzu, möglichst schnell und unkompliziert Anfragen an die Ontologie beantworten zu können.

6.3.1.2 Robustheit

In einer dynamischen Umgebung ist es wichtig, dass die Ontologie trotz aller Änderungen stets konsistent bleibt. Hierfür ist es unerlässlich, die Einhaltung aller *Restriktionen* und Vorgaben zu überprüfen. Während der **Entwicklung** ist die *Open World Assumption* von *OWL* sehr angemessen, erlaubt sie doch, mittels *Reasonern* eine automatisierte Klassifikation (vergl. [Knublauch et al. 04] S. 69 bzw. [Antoniou & Harmelen 04] S. 145). In der **Laufzeit** ist ein solches Verhalten jedoch eher hinderlich, da es zu ungewünschten Schlussfolgerungen führen kann. Weiterhin wird eine Verletzung einer *Restriktion* normalerweise nicht als Fehlerfall, sondern als implizite Erweiterung der Ontologie interpretiert. Dies ist in der Laufzeit nicht akzeptabel, da dies zu ungewünschten Nebeneffekten führen kann, die die Funktionsfähigkeit des Systems beeinträchtigen (vergl. [Knublauch et al. 04] S. 35).

> Beispiel:
> Würde ein Gerät im *OrganicRoom* angeben, es stünde an dem Ort (*Location*) *music*, so würde in der Open-World-Assumption angenommen, dass ein Ort auch eine Komponente, wie es *music* ist, sein kann. Ganz offensichtlich würde dies die Ontologie des Raumes sehr unerwünscht verfälschen.

Aus diesen Gründen stellt die **Embedded Ontology Engine** sicher, dass alle *Restriktionen* exakt eingehalten werden indem sie einen Fehler meldet, sobald diese verletzt werden.

6.3.1.3 Dynamisches Verteilen von Wissen

Die größte Neuerung durch den *OrganicRoom* ist, dass Wissen nicht nur an einem Ort gesammelt und verarbeitet wird, sondern zwischen Geräten dynamisch verteilt werden muss. *OWL* spezifiziert zwar einen Import, dieser erlaubt jedoch immer nur eine vollständige Ontologie als Erweiterung der bestehenden zu importieren (vergl. [Knublauch et al. 04] S. 144). Dies ist in diesem Szenario nicht sinnvoll, da alle Geräte auf einem gemeinsamen Modell fußen und somit immer schon einen gewissen Teil der Ontologie kennen werden. Daraus folgt, dass ontologische Stränge zusammengeführt und auf ihre Vereinbarkeit geprüft werden müssen, um Widersprüche zu vermeiden.

Daher wird nur das minimal notwendige Wissen verbreitet. Minimal verteilt heißt, das kein Gerät mehr als die zur Konsistenz nötigen Daten in seiner Ontologie besitzt.

Treffen sich zwei Geräte, so wird also zunächst nur die *Instanz*, die das Gerät ontologisch beschreibt, ausgetauscht. Daher benötigt die **Embedded Ontology Engine** für den Fall eine Methodik, dass unbekannte Elemente referenziert werden, um gezielt weitere Informationen von dem Gerät nachzufragen, das dem aktuellen Informationsfluss zugeordnet ist. Die **Embedded Ontology Engine** besitzt hierfür das *Retriever*konzept.

Hierbei zeigt sich auch ein weiteres Problem der *OWL-XML* Repräsentation. Wie aus der Syntax ersichtlich (vergl. Kapitel A2.1.3), werden bei der *Klassenbeschreibung* nicht die sich auf die Klasse beziehenden *Properties* erwähnt. Diese zu kennen ist für die Überprüfung empfangener *Instanzen* aber unabdingbar notwendig. Von daher ist es sinnvoll, an dieser Stelle die Syntax der *Klassen* von *OWL* entsprechend zu erweitern. So lässt sich nach empfangener *Instanz* die *Klasse* nachfragen, falls sie nicht bekannt ist, und es werden sofort die durch sie benutzten *Properties* mitgeteilt. Dies ermöglicht, die Gültigkeit der *Instanz* zu überprüfen.

6.3.1.4 Schnittstellen zu Standardtools

Da *OWL-XML* für das Anwendungsgebiet eher ungeeignet ist, muss, um weiterhin Interoperabilität und Zugriff auf Tools wie *Reasoner* zu gewährleisten und um zu vermeiden, eigene Werkzeuge zur Entwicklung von Ontologien entwickeln zu müssen, eine Möglichkeit geschaffen werden, Ontologien zwischen *OWL-XML* und der von der hier vorgestellten **Embedded Ontology** verwendeten Repräsentation umzuwandeln.

Hierzu bietet sich das **Protégé OWL Plugin** als Schnittstelle an.

6.3.2 Aufbau

Die **Ontology Engine** basiert auf der Klasse **OntologyManger**. Diese verwaltet die gesamte Ontologie und bietet Funktionen, die Ontologie zu erzeugen, zu erweitern und zu durchsuchen. Die Ontologie selbst wird durch die Klassen *OWLClass, OWLIndividual* und *OWLProperty* gespeichert. Diese repräsentieren die durch *OWL* spezifizierte Wissensdatenbank.

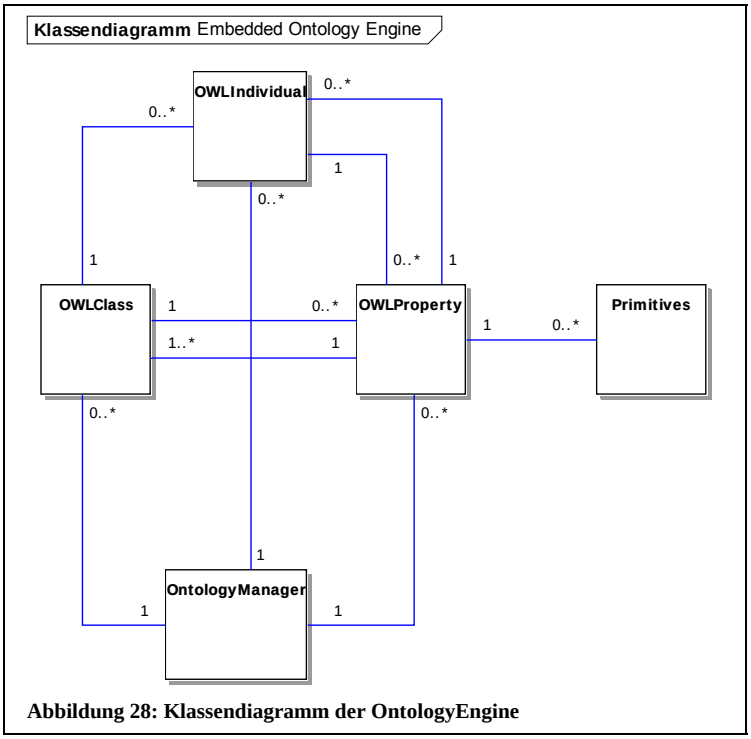


Abbildung 28: Klassendiagramm der OntologyEngine

Weiterhin stellt der **OntologyManager** Methoden zum Verteilen des Wissens zur Verfügung, durch die Elemente der Ontologie serialisiert und somit über das Netz übertragbar werden.

Die Klassen *OWLClass, OWLIndividual* und *OWLProperty* speichern die ontologischen Elemente und überprüfen gleichzeitig ihre Gültigkeit. Weiterhin bieten auch sie Methoden zum Durchsuchen und zum Serialisieren.

Zur Verdeutlichung das Beispiel aus Kapitel 4.2 mit der **Ontology Engine** umgesetzt:

```
01: OntologyManager owlModel = new OntologyManager();
02:
03: OWLClass AutonomicUnit =
04:         owlModel.createOWLClass("AutonomicUnit");
05:
06: OWLProperty PowerConsumption =
07:         owlModel.createOWLProperty("PowerConsumption", 1 );
08: AutonomicUnit.addProperty(PowerConsumption);
09:
10: OWLProperty has = owlModel.createOWLProperty("has",
11:                               owlModel.findClass("Component", null);
12: AutonomicUnit.addProperty(has);
13:
14: OWLIndividual Radio = owlModel.createOWLIndividual("Radio",
15                                                    AutonomicUnit);
16: Radio.addProperty(has, owlModel.findIndividual("Musik", null));
```

Listing 11: Erzeugen der Klasse AutonomicUnit mit der Embedded Ontology Engine

In Zeile 1 wird eine neue Ontologie angelegt. Daraufhin kann in Zeile 3 die erste *OWL Klasse* erzeugt werden. In den Zeilen 6-8 wird die *Property* **PowerConsumption** angelegt. Im Unterschied zu **Protégé** wird nicht zwischen *Object-* und *DatatypeProperty* unterschieden, da dieser Unterschied anhand der *Range* festgemacht werden kann. Die *Range* wird bei dem Befehl als zweiter Parameter übergeben. Die **Ontology Engine** speichert den Typ des übergebenen Objektes, in diesem Falle 1, also Integer. Die *Domain* dieser *Property* wird nun implizit gesetzt, indem die *Property* der Klasse **AutonomicUnit** in Zeile 8 hinzugefügt wird.

In den Zeilen 10-12 wird die *Property* **has** angelegt. Sie verbindet ein **AutonomicUnit** mit einer oder mehreren *Instanzen* der *Klasse* **Component**. Entsprechend ist sie eine *ObjectProperty*, was die **OntologyEngine** anhand des Parameters für die *Range* feststellt. In diesem Beispiel gehen wir an dieser Stelle davon aus, dass die *Klasse* **Component** bereits erzeugt wurde, was ermöglicht zu zeigen, wie man die Ontologie nach vorhandenen Objekten durchsuchen kann. Dies erledigt die Methode *findClass()*. Als zweiten Parameter kann ihr eine *OWLRetriever*-Instanz übergeben werden. Diese wird benutzt, falls die gesuchte *Klasse* nicht in der Ontologie enthalten ist, um zu versuchen, sie anzufordern. Danach wird der *Property* die *Klasse* **AutonomicUnit** als Domain hinzugefügt.

Abschließend wird in der Zeile 14 die *Instanz* **Radio** erzeugt. Da ein Radio Musik erzeugt, wird das **Radio** in Zeile 16 mit der *Property* **has** mit der *Instanz* **Musik** der *Klasse* **Component** verbunden.

Die Arbeitsweise der **OntologyEngine** auf dieser Ebene ist also recht ähnlich der des **Protégé OWL Plugins**, unterscheidet sich jedoch im Detail etwas, so muss nicht mehr manuell zwischen *ObjectProperty* und *DatatypeProperty* unterschieden werden, was den Umgang mit *Properties* vereinfacht. Auch wurde die aus der objektorientierten Programmierung gewohntere Herangehensweise gewählt, dass *Properties* zu *Klassen*

assoziiert werden, indem die Methode *addProperty()* aufgerufen wird, anstatt ein *setDomain()* auf die *Property* auszuführen.

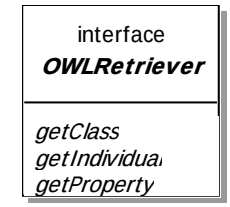


Abbildung 29: Klassendiagramm OWLRetrieer

Größter Unterschied für den Anwender dürfte jedoch das Vorhandensein des *OWLRetrievers* sein, der bei jeder Abfrage an die Ontologie mit angegeben werden muss. Der *Retriever* implementiert drei Methoden, die aufgerufen werden, wenn ein Element der Ontologie nachgefragt wurde, das in der Ontologie derzeit nicht vorhanden ist. Aufgabe dieser Methoden ist es dann, diese Elemente zu ermitteln und als Rückgabewert zurückzuliefern oder einen Fehler zu melden, falls diese auch so nicht beschafft werden können. In der Implementierung des *OrganicRoom* wird hierbei beispielsweise über das **L3 Netz** an den Peer, mit dem in diesem Kontext gerade kommuniziert wird, eine Anfrage geschickt. Sobald eine Antwort erhalten wurde, wird das entsprechende Element über die S*erialisierung* in die Ontologie eingefügt und dann weiter fortgefahren.

6.3.3 Arbeitsweise

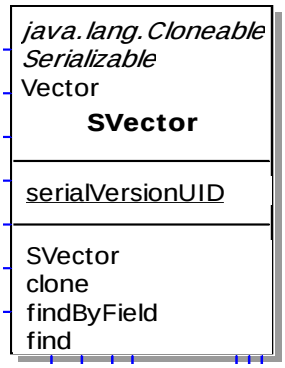


Abbildung 30: Klassendiagramm SVector

Die **Engine** leistet vor allem drei Dinge: Sie **speichert** die Ontologie, sie ermöglicht, die Ontologie gezielt nach Fragestellungen zu **durchsuchen** und schließlich **überprüft** sie die Ontologie auf Konsistenz und semantische Korrektheit.

Das Fundament ist hierbei eine Erweiterung des Java *Vectors*, genannt *SVector* nach „Searchable Vector“. Mit ihm werden alle Mengen gespeichert, wie z.B. in dem OntologyManager alle *Klassen, Individuals, Properties* sowie in den *Klassen* selbst die Menge aller zu ihr gehörenden *Properties* oder in den *Individuals* die Menge der *Properties* samt ihrer *Values* etc. Der *SVector* ermöglicht es, die in ihm gespeicherte Menge nach einem Feld mit einem Wert zu durchsuchen.

Zur Demonstration hier ein Stück Beispielcode. Aus der Liste aller *Individuals* soll eines gesucht werden, das zur *Klasse* **AutonomicUnit** gehört.

```
01: OWLIndividual aUnit =
02:  owlModel.getAllIndividuals.findByField(AutonomicUnit, "heritage")
```
Listing 12: Beispiel für SVector

Die Methode *findByField()* durchsucht den *SVector* und vergleicht, ob der *Member* **heritage**[12] eines enthaltenen Objektes auf die *Klasse* **AutonomicUnit** verweist.

Über diese mächtige und flexible Suchfunktionalität werden sämtliche Abfragen an die Ontologie abgewickelt.

[12] **Heritage** verweist auf die *Klasse*, zu der ein *Individual* gehört.

Weiterhin ist die Semantik zu Überprüfen, wie in Kapitel 0 definiert. An erster Stelle bedeutet das, die Existenz referenzierter Elemente zu überprüfen. Da in Java nur auf existierende Objekte verwiesen werden kann, ist dies relativ leicht sicherzustellen. Schwieriger ist es schon sicherzustellen, dass *Restriktionen* eingehalten werden. Das bedeutet vor allem bei einem *Individual* sicherzustellen, dass nur *Properties* verwendet werden, die in der *Klasse* auch erlaubt sind. Weiterhin muss sichergestellt werden, dass die *Ranges* der Properties eingehalten werden. Dies ist bei *DatatypeProperties* noch recht simpel, bei *ObjectProperties* muss jedoch nicht nur sichergestellt werden, dass das Objekt vom richtigen Typ ist (also eine Instanz von *OWLIndividual*), sondern das *OWLIndividual* muss selbst von der in der *Range* spezifizierten *OWLClass* abstammen oder aber von einer *Subklasse* dieser spezifizierten *OWLClass*. Abbildung 31 zeigt diesen Vorgang.

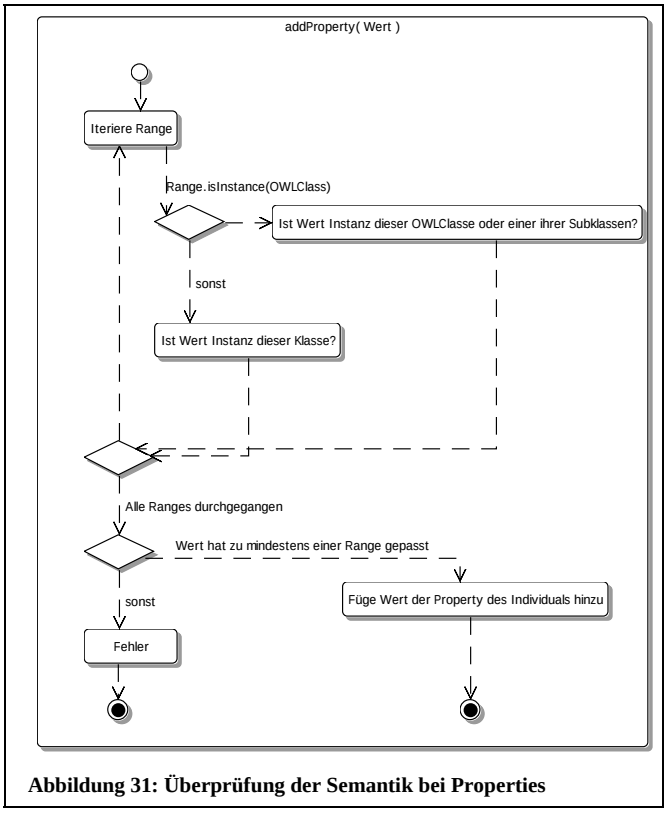


Abbildung 31: Überprüfung der Semantik bei Properties

Die exakte Arbeitsweise der **Ontology Engine** ist im Detail der Javadoc im Anhang, Kapitel A1, zu entnehmen. Das vollständige Klassendiagramm der **Ontology Engine** ist im Anhang, Kapitel A3, beigelegt.

6.3.4 Syntax der Ontology Engine

Wie bereits in Kapitel 6.3.1 motiviert, war es für das Szenario *OrganicRoom* aus Ressourcengründen sinnvoll, *OWL* nicht in seiner *RDF-XML* Repräsentation zu verwenden, was allerdings eine andere Syntax bedingt. Weiterhin hat es sich, wie dort erwähnt, auch als sinnig erweisen, die **abstrakte Syntax** von *OWL* leicht zu modifizieren. In diesem Kapitel soll die neue Syntax vorgestellt und die Abweichungen zur **abstrakten Syntax** gezeigt und begründet werden. Die Spezifikation der **abstrakten Syntax** ist vollständig in Anhang A2 enthalten.

Ziel der Engine war es, ein für das Szenario möglichst sinnvolles Subset von *OWL Lite* zu implementieren. Hierbei sollte ganz bewusst auf einige Elemente verzichtet werden, die in der **Laufzeitphase** des Systems entweder keine Rolle spielten oder den Ressourcenbedarf deutlich erhöht hätten ohne einen für dieses Anwendungsgebiet merklichen Mehrwert zu erbringen.

Allerdings sind diese Elemente zum Teil während der **Entwicklungsphase** sehr nützlich. Dadurch, dass hierfür **Protégé** verwendet wird und eine Schnittstelle zum Import und Export existiert, stellt dies jedoch keine wirkliche Einschränkung dar.

Weggefallene Elemente:

- Deprecated
  Die Möglichkeit, *Klassen* oder *Properties* als *deprecated* zu markieren, wurde nicht implementiert. Dies ist während der **Entwicklung** nützlich, um nicht mehr verwendete bzw. überholte Elemente zu kennzeichnen, und damit eine Migration zu neuen zu ermöglichen. Während der **Laufzeitphase** ist dies jedoch nicht sinnvoll.

- EquivalentClasses
- EquivalentProperties
  Die Begründung entspricht der obigen. *Klassen* bzw. *Properties* als äquivalent zu definieren ist in der **Entwicklung** ein wünschenswertes Element zur Organisation der Ontologie, in der **Laufzeitphase** ist diese Information jedoch redundant, äquivalente *Klassen* sollten zur Ressourcenersparnis auch wirklich durch eine einzelne *Klasse* ausgedrückt werden.

- Modality
  Da *Restriktionen* implizit implementiert sind (s.u.), entfällt dieses Element, das sonst erlaubt, Klassen über Restriktionen zu deklarieren.

- Properties: inverseFunctional, inverseOf, Symmetric, InverseFunctional, Transitive
  Auf diese Attribute wurde aus dem Grund der limitierten Ressourcen und der fehlenden Verwendungsmöglichkeiten im Szenario verzichtet, sie stellen jedoch eine interessante und leicht zu implementierende Erweiterung der **Ontolog Engine** dar.

Modifizierte Elemente:

- Restriction
  - *allValuesFrom*
    Diese *Restriktion* entspricht im wesentlichen der Definition einer *Range* bei einer *Property*, nur das sie im Gegensatz zu dieser eher als Verpflichtend angesehen wird (vergl. [Knublauch et al. 04] S. 35). Da in der **Ontology Engine** *Ranges* von *Properties* verpflichtend sind, ist dieses Element redundant.

  - *cardinality*
    OWL Lite unterstützt nur Kardinalitäten von 0 oder 1. Eine Kardinalität von 1 ist gleichbedeutend damit, eine *Property* als *Functional* zu definieren, d.h. sie kann nur einen Wert aufnehmen.

  - *someValuesFrom*
    Diese Restriktion ist nicht redundant. Da sie im Rahmen des Anwendungsgebietes allerdings keine Verwendung findet, wurde auf sie verzichtet, um Ressourcen zu sparen, die Implementierung stellt auch hier kein prinzipielles Problem dar.

- DatatypeProperty / ObjectProperty
  Da durch die *Range* einer *Property* definiert wird, ob eine *Property* einen Datentyp oder ein *OWL*-Objekt referenziert, wurde diese Unterscheidung bei der **Ontology Engine** nicht explizit ausgedrückt und *Datatype-* bzw. *ObjectProperties* werden beide als *Properties* behandelt.

Zur Veranschaulichung wird in Abbildung 32 die Repräsentation der in Kapitel 5.7 vorgestellten *Klasse* **Action** und zweier *Instanzen* in *OWL-XML* und in der der **Ontology Engine** gegenübergestellt:

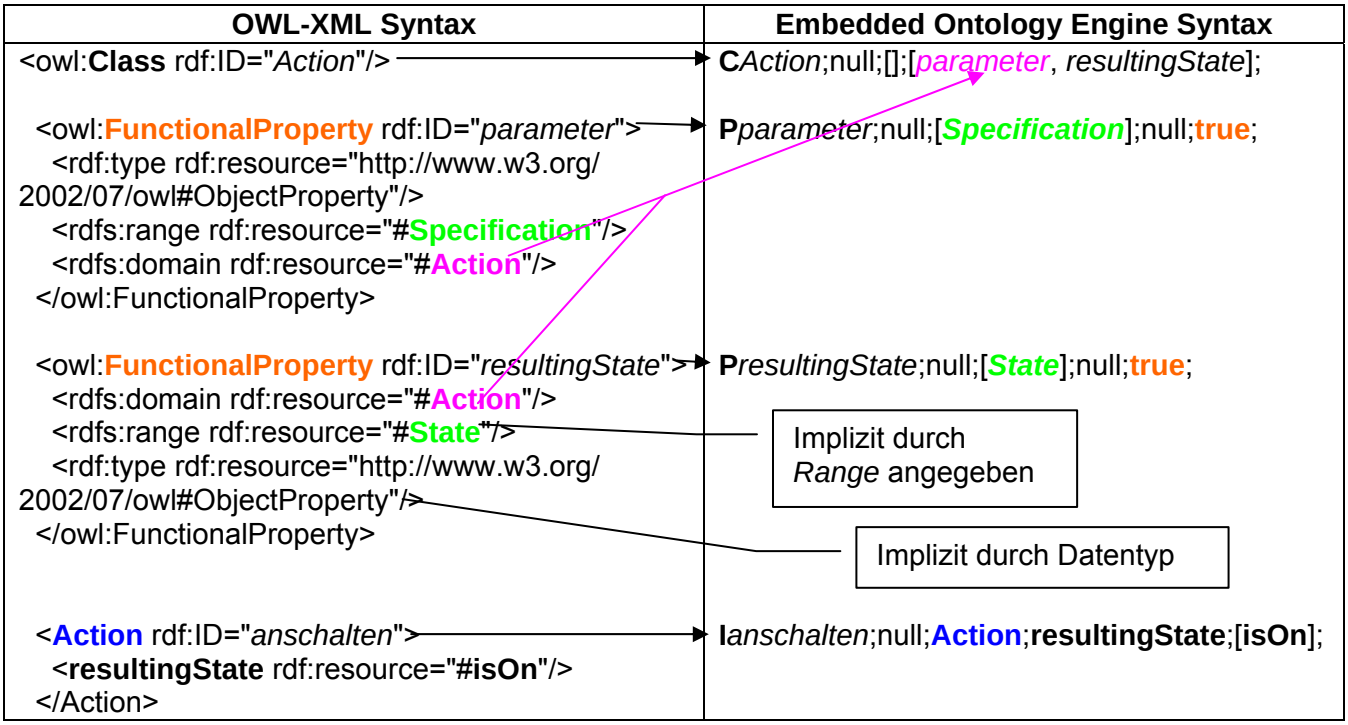


Abbildung 32: Beispiel Syntax OWL - Ontology Engine

Zum Abschluss folgt noch die konkrete Syntax in erweiterter Backus-Naur Form. Wie auch in Kapitel A2 gilt folgende Vereinbarung:

> Terminale stehen in Anführungszeichen, non-terminale in Fettdruck. Alternativen werden durch ein | angezeigt. Komponenten, die maximal einmal auftreten dürfen, werden in eckigen Klammern […] aufgezählt, solche, die beliebig oft auftreten dürfen, in geschweiften {…}.

Ebenso wie in der originalen Syntax[13] besteht die Ontologie weiterhin aus Axiomen und Fakten.

ontology ::= **axiom** | **fact**

Fakten sind die *Instanzen* bzw. *Individuals*.

fact ::= **individual**

individual ::= 'I' [**individualID**] ';' [**comment**] ';' [**type**] ';' { **value** } ';'

type ::= **classID**

value ::= **PropertyID** ';' **individualID** ';'
| **PropertyID** ';' **dataLiteral** ';'

Individuals bestehen aus einer ID, einem Kommentar, dem Typ, das ist die Klasse, von der sie abstammen, sowie ihren *Property/Wert*-Pärchen.

[13] Siehe Kapitel A2.1

Axiome bilden die *Klassen* und *Properties.*

axiom ::= 'C' [**classID**] ';' [**comment**] ';' [**super**] ';' { **propertyID** } ';'
super ::= **classID**

Klassen bestehen aus einer ID und einem Kommentar, dann der Referenz auf ihre Superklasse und (im Unterschied zum normalen *OWL* Syntax) aus der Referenz auf ihre *Properties*.

axiom ::= 'P' [**propertyID**] ';' [**comment**] ';' { **range** } ';' [**parent**] ';' [**Functional**] ';'

range ::= **classID** | **dataLiteral**
parent ::= **propertyID**
Functional ::= 'true'
| 'false'

Properties bestehen aus einer ID und einem Kommentar, dann ihren *Ranges,* der Referenz auf ihre *Superproperty* (parent) und der Angabe, ob sie *Functional* sind oder nicht. Eine *Functional Property* kann lediglich einen Wert referenzieren. Der große Unterschied hierbei ist, wie erwähnt, die fehlende explizite Unterscheidung in *Datatype-* und *ObjectProperties.*

6.4 Der OWL-XML Importer / Exporter

Der Importer stellt die Schnittstelle zwischen **Protégé** und damit *OWL-XML* und der **Embedded Ontology Engine** dar. Aufgabe ist es, mit dem bereits in Kapitel 4.2 vorgestellten **Protégé OWL Plugin** eine Ontologie einzulesen und sie dann schrittweise mit der **Ontology Engine** zu konvertieren und das Ergebnis serialisiert abzuspeichern.

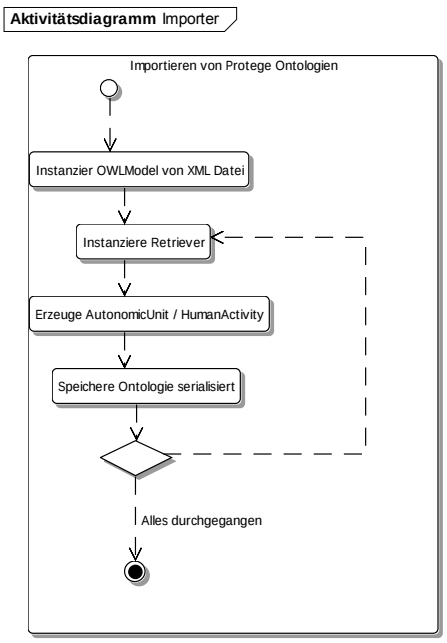


Abbildung 33: Aktivitätsdiagramm des Importers

Hierbei spielt der *Retriever* eine zentrale Rolle. Der Importer implementiert im wesentlichen einen *Retriever*, der die fehlenden Daten mittels dem **OWL Plugin** erzeugt. Zum Konvertieren reicht es dann, in der Hauptroutine einfach die *Klasse* **HumanActivity** zu iterieren. Durch den *Retriever* werden so automatisch alle notwendigen Elemente in der **Embedded Ontology** erzeugt. Diese wird danach serialisiert gespeichert und ist somit für die **Organic Room Software** einsatzbereit.

Die Aktivitätsdiagramme der einzelnen Klassen des *Retrievers* finden sich im Anhang A4.

Der **Exporter** verhält sich hierzu ganz analog. Da jedoch das **Protégé OWL Plugin** nicht über ein *Retriever*konzept verfügt wird dieses nachgebildet. Das heißt, dass es auch hier die drei *Klassen* **createClass**, **createIndividual** und **createProperty** gibt, wenn diese aber ein Element benötigen, das nicht existiert, so rufen sie sich entsprechend gegenseitig auf. Abschließend wird die Ontologie mittels dem **Protégé OWL Plugin** im *OWL-XML* Format abgespeichert und ist somit wieder mit **Protégé** bearbeitbar.

6.5 Die Organic Room Software

Die *Organic Room Software* hat die Aufgabe, die entwickelte *Organic Room* Ontologie zum Einsatz zu bringen. Sie besteht aus zwei Teilen, einmal dem System, das auf der **Embedded** Seite auf dem **XC167** läuft, sowie dem Teil für den **PDA**.

Wie eingangs erwähnt, dient die **Embedded** Seite als eine Art Zwischenstecker. Ihre Aufgabe ist es, eine ontologische Beschreibung des Gerätes, das mit ihr verbunden ist, zur Verfügung zu stellen sowie dieses Gerät zu steuern.

Die **PDA** Seite sammelt diese Informationen und präsentiert sie dem Benutzer in übersichtlicher Form. Dieser kann darauf zentral alle Geräte des Raumes entweder gezielt steuern oder Aktivitäten auswählen, wobei daraufhin die Software versucht, mit Hilfe der Ontologie für diese Aktivitäten passende Geräte zu finden und passende Aktionen für diese zu ermitteln. Der Benutzer kann diese Vorschläge entweder direkt annehmen oder modifizieren. Diese Auswahl wird gespeichert und somit ein zu dem Verhalten des Nutzers passendes Profil auf dem **PDA** angelegt[14]. Schließlich zeigt die Software noch den aktuellen **Energieverbrauch** des Nutzers an.

Die Arbeitsweise der Software gliedert sich in zwei Phasen. Zunächst gilt es für das **PDA** alle Geräte im Raum zu finden und in seine Ontologie zu integrieren. Danach kann diese Information dem Benutzer angezeigt werden und dieser kann mit den Geräten interagieren.

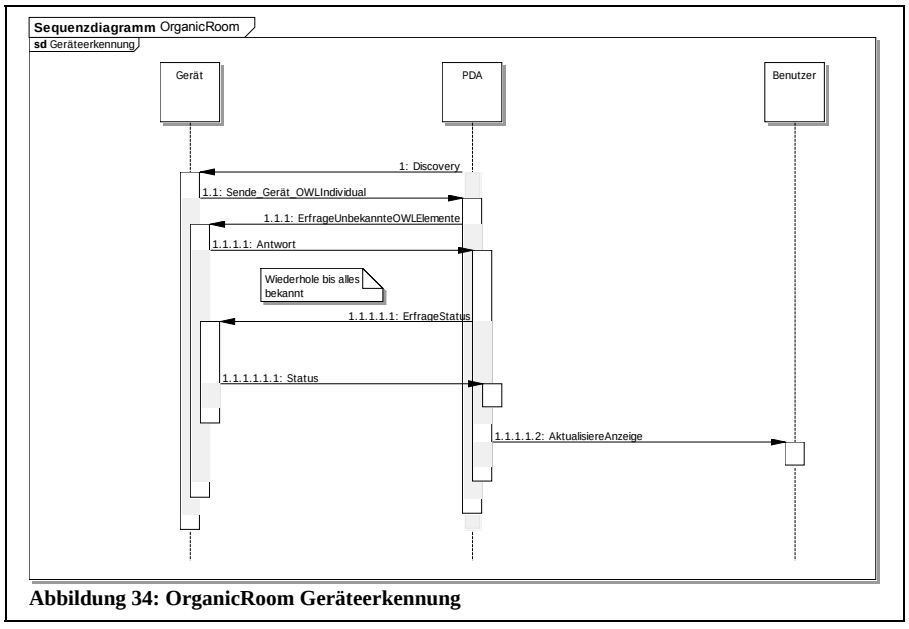


Abbildung 34: OrganicRoom Geräteerkennung

[14] Hierbei ist anzumerken, dass dieses Profil ausschließlich auf dem PDA des Nutzers gespeichert ist und nicht abgerufen werden kann oder anderweitig verbreitet wird. Auch können die Geräte den Nutzer nicht identifizieren. Die Privatsphäre des Nutzers bleibt also in jedem Fall gewährleistet und dieses System kann nicht von Dritten dazu benutzt werden, Menschen zu überwachen.

Das Finden der Geräte funktioniert über den *Discoveryservice* des **L3** Protokolls. Dieses sendet periodisch Anfragen an seine Umgebung, um neue Geräte zu identifizieren. Wurde ein anderes Gerät gefunden, auf dem auch der *OntologyService* läuft, so übermittelt dieses seine ontologische Beschreibung. Im Falle des *OrganicRooms* ist das eine *Instanz* der *Klasse* **AutonomicUnit**, die das jeweilige Gerät beschreibt. Werden bei der Integration in die Ontologie des **PDA**s unbekannte Elemente referenziert, so wird über den *Retriever* bei dem Gerät nach diesen gefragt. Kennt das Gerät dieses Element selbst nicht, so kann es auch an einen anderen Peer verweisen. Dies ermöglicht ein **Clustering**, d.h. es kann im Raum eine für diesen zentrale Instanz geben, die für kleinere Geräte ontologische Informationen speichert und so diese entlastet.

Abbildung 35: Organic Room Software Hauptbildschirm

Nachdem das gefundene Gerät vollständig in die Ontologie integriert wurde, wird der Status des Gerätes erfragt (z.B. **an**/**aus** bei **Lampen**) und die Anzeige für den Benutzer entsprechend aktualisiert. Dieser kann jetzt entweder ein Gerät direkt oder eine Aktion auswählen.

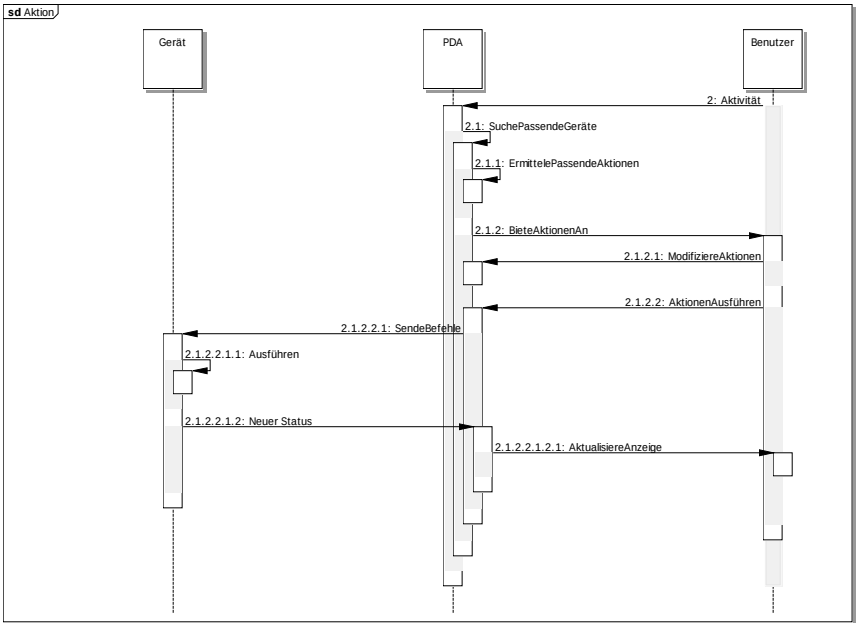


Abbildung 36: OrganicRoom Aktion ausführen

Wählt der Nutzer eine Aktion, so versucht der **PDA** zu den Attributen der Aktion passende Geräte zu finden und für diese Geräte passende Befehle. Diese bietet er dem Benutzer an, der daraufhin noch Feinabstimmungen vornehmen kann.

So wird beispielsweise die Aktion **Entspannen** über die Attribute **RecreativeUnit**, **Wohnbereich**, **warm** und **gedimmt** definiert. Die Software zeigt die gefundenen Geräte an (erste Spalte), daraufhin die passende Komponente mit dem dazugehörigen Attribut (zweite Spalte). In der letzten Spalte befindet sich je ein Button mit der vorgeschlagenen Aktion. Falls der Benutzer diese Aktivität schon einmal in diesem Raum und damit auch mit diesem Geräten ausgeführt und veranlasst hat, den gewählten Befehl zu speichern, wird stattdessen der gespeicherte angezeigt. Ansonsten wird die Standardaktion (vergl. Kapitel 5.6) ermittelt.

Abbildung 37: Organic Room Aktion "Entspannen"

Durch Druck auf den Button kann diese Aktion wie gewünscht geändert werden, siehe Abbildung 43 für das Beispiel **EntertainmentCenter**. Es werden für jede **Component** alle verfügbaren Aktionen aufgelistet. Will man jetzt nicht irgendwelche Musik, sondern einen bestimmten Typ starten, so lässt sich dieses über den Button **start** entsprechend einstellen. Im Beispiel Abbildung 38 wurde hier **Bluesrock** ausgewählt. Bei der letzten Aktion traf die Wahl Bachs Orgelwerke, was sich die Software auf Anweisung gemerkt hat. Alternative Befehle hätte mit **listContent** eine genaue Auflistung der vorhandenen Stücke von dem Gerät angefordert werden können, worauf ein konkretes Stück hätte ausgewählt werden können. Abbildung 39 zeigt das Ergebnis, wobei sich hier der Benutzer noch entschieden hat, das ihm eine Raumtemperatur von 25°C lieber ist.

Abschließend wählt der Benutzer „Starten“ oder „Starten und merken“, woraufhin die Software die entsprechenden Befehle sendet. Für das **EntertainmenCenter** ist der Befehl z.B.

```
amusic:start:T:Bluesrock
```
Listing 13: Befehl an EntertainmentCenter

Das Ergebnis der Befehle ist in Abbildung 40 zu sehen. Die entsprechenden Geräte wurden eingeschaltet, was zur Übersichtlichkeit durch blaues Hervorheben gekennzeichnet ist. Weiterhin wird der Gesamtenergieverbrauch berechnet und angezeigt.

Abbildung 43: Optionen des Gerätes EntertainmentCenter

Abbildung 38: Auswahl eines Musiktypes

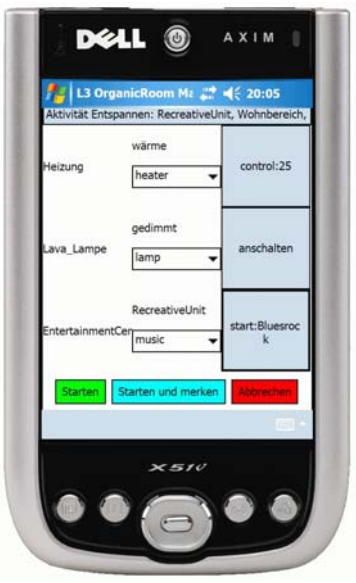


Abbildung 39: Parametrierte Aktion "Entspannen"

Abbildung 40: OrganicRoom nach Ausführen der Aktion "Entspannen"

Abbildung 41: Erstellen der Aktivität "Lesen"

Abbildung 42: OrganicRoom mit neuer Aktivität "Lesen"

Ein direktes Selektieren eines Gerätes bringt einen Dialog wie in Abbildung 43 hervor. Weiterhin ist es auch möglich, neue Aktivitäten dynamisch zu erstellen. In Abbildung 41 wird die in Kapitel 5.1 vorgestellte Aktivität **Lesen** mit den dort definierten Attributen erstellt. Diese wird dann der Liste der Aktivitäten hinzugefügt, siehe Abbildung 42.

Außerdem können noch Aktivitäten gelöscht oder alle Geräte auf einmal ausgeschaltet werden. Hierbei findet die Software für jedes Gerät den passenden Befehl zum Ausschalten automatisch. Sie schaut sich den Ergebniszustand der Standardaktion an und findet über die *Property* **inverseOf** hierzu das logische Gegenteil. Ein Matching ermittelt hierzu die passende **Action**, die diesen Zustand erzeugt. Daraufhin wird dieser Befehl ausgeführt (vergl. Kapitel 5.7).

6.6 Das Matching im Detail

Das Matching stellt die zentrale Komponente der *Organic Room Software* dar. Sie ermittelt zu *Aktivitäten* passende Geräte. Jede *Aktivität* ist durch eine Reihe *Attribute* definiert. Beim *Matching* müssen alle Geräte gefunden werden, die eines dieser Attribute erfüllen und am gewünschten Ort sind.

Die folgende Abbildung verdeutlicht den Vorgang:

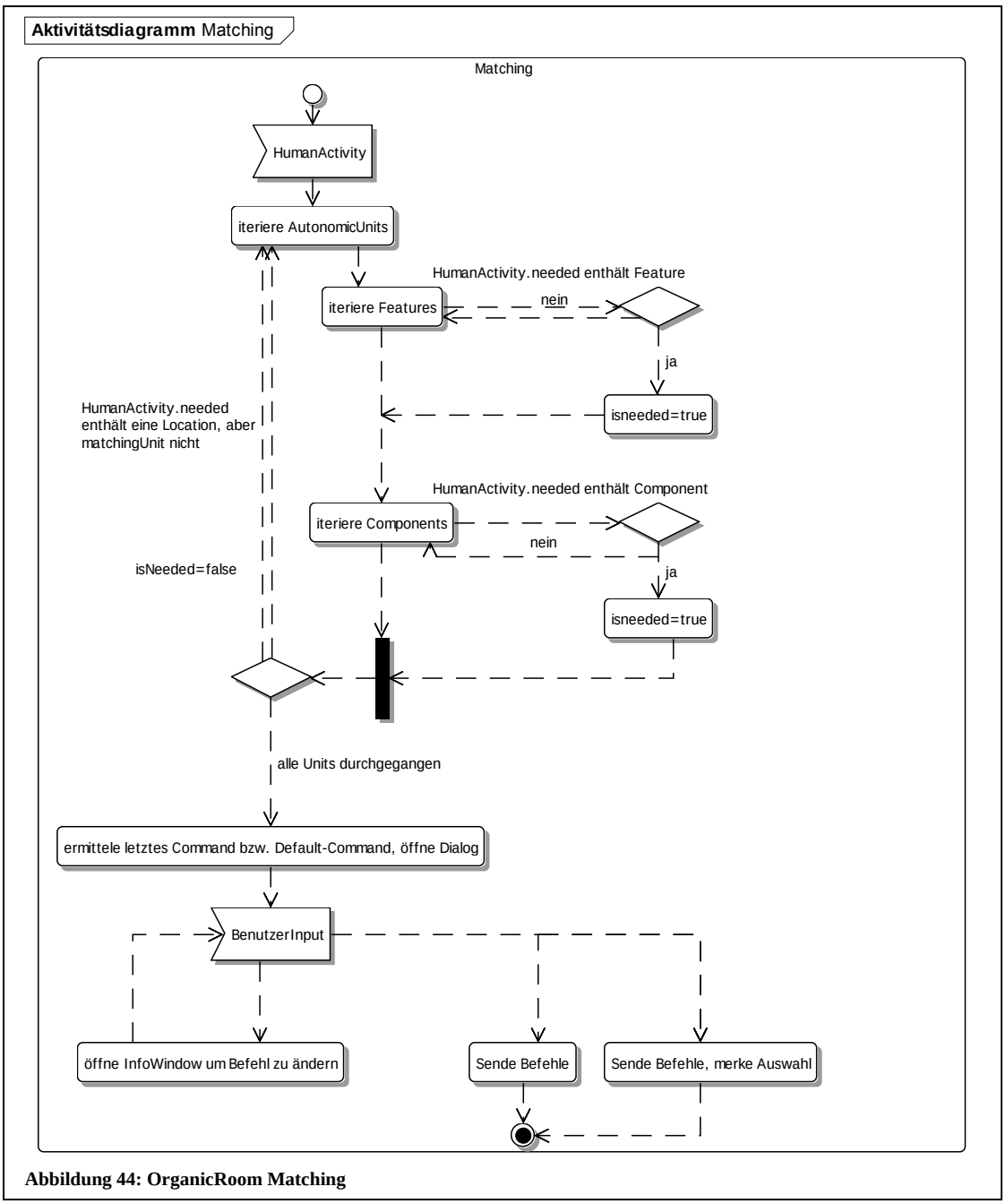


Abbildung 44: OrganicRoom Matching

6.7 Die embedded Version

Ziel der Konzeption des **Embedded Ontology Engine** war, die Anforderungen an den eigentlichen Embedded-Teil möglichst klein zu halten, um eine möglichst kleine Plattform benutzen zu können. Mit dem vorgestellten Konzept muss die embedded Version minimal lediglich die Serialisierung des *Individuals* speichern, das sie selbst beschreibt. In diesem Falle würden alle weiteren Anfragen per *Redirection* an einen anderen Peer weitergeleitet werden. Außerdem muss die Software nur noch Befehle

entgegen nehmen können, um das Gerät zu steuern, und um den Status mitteilen können. Listing 14 zeigt als Beispiel die Selbstbeschreibung der **Lampe**.

```
// Ontology for Lampe_Arbeitsplatz
const char selfdescription[] =
  "meLampe_Arbeitsplatz@%02X%02X%02X%02X%02X%02X%02X%02X;null;
  AutonomicUnit;at;[Arbeitsplatz];PowerConsumption;[60];description;
  [hell];has;[lamp];";
```

Listing 14: Beschreibung des Gerätes LavaLampe in der Embedded Version

In einer etwas größeren Variante benötigt die Embedded Version eine vollständige Ontologie. Diese wird in ebenso serialisierter Form in einer Liste abgelegt und bei Anfragen entsprechend durchsucht.

```
struct OntData { const char *name; const char *desc; };

struct OntData Ontology[] = {
      {"PowerConsumption", "PPowerConsumption;null;[1];null;true;"},
      ...
      {"", ""}
};
```

Listing 15: Struktur der embedded Version der Ontologie

Weiterhin muss sie natürlich das Kommunikationsprotokoll unterstützen.

6.8 Kommunikationsprotokoll

Im Folgenden wird das Kommunikationsprotokoll spezifiziert, mit dem Peers ihre ontologischen Daten austauschen können.

Anfrage der Selbstbeschreibung:
Wird ein Peer neu entdeckt, so muss für seine ontologische Identifikation die Selbstbeschreibung bekannt sein. Der Befehl, diese Anzufordern ist:

Command ::= 'w'

Die Antwort mit der serialisierten Beschreibung ist:

Command ::= 'me' [**Individual**]

Anfrage nach ontologischen Elementen:
Wenn die Selbstbeschreibung auf unbekannte Elemente verweist, so müssen diese nachgefragt werden. Die entsprechenden Befehle lauten:

Command ::= 'i' [**Individual-Name**]
| 'c' [**Klassen-Name**]
| 'p' [**Property-Name**]

Die Antworten hierzu sind:

Command ::= 'I' [**Individual**]
| 'C' [**Klassen**]
| 'P' [**Property**]

Wurde nach einem dem Gerät unbekannten Element gefragt, ist die Serialisierung der Antwort entsprechend leer. Alternativ kann die Anfrage an einen anderen Peer umgeleitet werden.

Die Syntax hierbei ist:

Command ::= 'R' [**Elementname**] ':' [**UUID**]

Statusabfrage:
Um den aktuellen Status einer *Komponente* des Geräts zu erfragen wird folgender Befehl verwandt:

Command ::= 's' [**Individual**]

Die Antwort ist:

Command ::= 'S' [**Statusbeschreibung**]

Aktion:
Mit Aktionen werden Geräte gesteuert. Eine *Aktion* wird immer auf eine *Komponente* bezogen und kann einen *Parameter* haben. *Parameter* können sich wiederum auf eine *Spezifikation* beziehen oder auf einen erfragten Wert.

Command ::= 'a' [**component**] ':' [**action**]

| 'a' [**component**] ':' [**action**] ':' 'T ':' [**Specification**]

| 'a' [**component**] ':' [**action**] ':' 'S ':' [**Wert**]

Beispiele für Aktionen sind:

```
alamp:anschalten
amusic:start:T:Bluesrock
amusic:listContent
amusic:start:S:Bach: Orgelwerke
```

Alle diese Aktionen führen zu einem Statusupdate des Gerätes. Lediglich die *Aktion* **listContent** stellt eine Besonderheit dar, da hierbei Informationen zurückgesendet werden. Das Format ist:

Command ::= 'L' [**component**] ':' { **Specification** } { Wert } ';'

Beispiel:

```
Lmusic:Klassik;Beethoven: Klaviersonaten;Klassik;Bach:
      Orgelwerke;Pop;Mamas&Papas: California Dreamin;
      ...
```

Diese Werte können wieder für Aktionen genutzt werden.

Mit diesem auf das Wesentliche reduzierten Protokoll ist ein vollständiges Verteilen von Ontologien möglich sowie die flexible Steuerung von Geräten.

6.9 Zusammenfassung

In diesem Kapitel wurden die für das Szenario *Organic Room* verwendeten Plattformen und Netzwerktechniken vorgestellt. Schließlich wurde die **Embedded Ontology Engine** mit ihren Anforderungen und Besonderheiten eingeführt und ihr Aufbau und ihre Arbeitsweise detailliert erläutert.

Weiterhin wurde die *Organic Room Software*, die den Raum quasi zum Leben bringt, detailliert erläutert.

6.10 Literaturempfehlung

Das Betriebssystem, auf dem der **VISE TCP/IP** Stack basiert, ist ausführlich in [Welge 01] beschrieben. Weiterhin sei auf die JavaDocs zu **VISE** sowie zur *OrganicRoom* Software verwiesen.

7. Zusammenfassung und Ausblick

Diese Arbeit hat in die Ontologie und ihre Anwendung im Bereich intelligentes Wohnen bzw. *Organic Room* eingeführt. Es wurde eine Ontologie für diesen Raum und eine für verteilte Kleinstsysteme geeignete Engine sowie Steuersoftware entwickelt. Hiermit ist es möglich, Geräte im Haus automatisch zu erfassen und gebündelt zu steuern. Weiterhin ist es möglich, Aktivitäten zu definieren, für die automatisch passende Geräte identifiziert und entsprechend parametriert werden.

Die geschaffene **Embedded Ontology Engine** ist eine angepasste Umsetzung von *OWL* für verteilte Systeme und implementiert alle notwendigen Features für verteilte Ontologien.

In Zukunft wird es interessant sein, die Fähigkeiten des ontologischen Modells und der Engine weiter auszubauen und in größerem Rahmen einzusetzen sowie weitere Geräte zu implementieren.

Eine für die Laufzeit interessante Erweiterung der **Engine** wird in [Horrocks et al. 00] vorgeschlagen, nämlich das *Reasoning* nicht auf die TBox (*Klassen* etc.) zu beschränken, sondern auf die ABox (*Instanzen*) zu erweitern sowie Regeln zur Definition zuzulassen, wie sie auch in [Mahmoudi & Müller-Schloer 06] verwendet werden. Dies ermöglicht, Attribute zu individualisieren. Nicht jeder wird das gleiche unter einem **hellen** Licht oder einer **langsamen** Musik oder einer **warmen** Temperatur verstehen. Hier ließen sich dann Regeln definieren, die sagen bei welcher Strahlleistung eine Lampe als **hell** einzustufen ist, welche BPM Zahl zu der Kategorie **langsam** führt oder welche Gradzahl **warm** zur Folge hat.

Ein weiteres Gebiet ist, eine Methodik zu definieren, die das Verhalten bei mehreren Nutzern im Raum behandelt. Optimalerweise sollten die Aktivitäten verschiedener Benutzer vom Raum unterstützt werden, was aber gerade bei gegenteiligen Bedürfnissen nicht ohne weitere Abstimmung möglich ist.

Abschließend ist zu überlegen, die Ontologie, die bisher als Dienst in der **L3** Middleware implementiert ist, tiefer zu integrieren und zu einem Bestandteil des Discoverydienstes zu machen. Für **JXTA** wurde dies in [Elenius & Ingmarrson 04] vorgeschlagen und für L3 wäre es in ähnlicher Weise als generischer Dienst ebenso sinnvoll. Die **Ontology Engine** stellt hierfür alles Benötigte zur Verfügung.

8. Verzeichnisse

8.1 Literaturverzeichnis

[Abel 04]
Kontextbasierte Informationsbereitstellung auf Basis des Semantic Web, Bachelorarbeit Fabian Abel, Universität Hannover, 2004

[Antoniou & Harmelen 04]
A Semantic Web Primer, Grigoris Antoniou and Frank van Harmelen, MIT Press 2004

[Broy et al. 00]
Broy, M.; Hegering, H.-G.; Picot, A.,: Integrierte Gebäudesysteme – Technologien, Sicherheit und Märkte. SecuMedia Verlag, Ingelheim 2000

[Elenius & Ingmarrson 04]
Ontology-based Service Discovery in P2P Networks, Daniel Elenius, Magnus Ingmarsson, 2004

[Gruber93]
Toward Principles for the Design of Ontologies Used for Knowledge Sharing, Thomas R. Gruber, 1993

[Halepovic & Deters 03]
The Costs of Using JXTA, Emir Halepovic, Ralph Deters, 2003

[Hendler 01]
Agents and the Semantic Web, James Hendler, 2001

[Horrocks et al. 00]
Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic SHIQ. In David McAllester, editor, Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000), volume 1831 of Lecture Notes in Computer Science, pages 482-496. Springer, 2000.

[Horrocks et al. 03a]
From SHIQ and RDF to OWL: The Making of a Web Ontology Language, Ian Horrocks, Peter F. Patel-Schneider, Frank van Harmelen, 2003

[IEA 05]
Inernational Energy Agency. Key World Energy Statistics, 2005

[Knublauch 04]
Editing OWL Ontologies with Protégé, Slides, Holger Knublauch, 2004
http://protege.stanford.edu/plugins/owl/publications/2004-07-06-OWL-Tutorial.ppt

[Knublauch 05]
The Protégé-OWL API - Programmer's Guide, Holger Knublauch, 2005, http://protege.stanford.edu/plugins/owl/api/guide.html

[Knublauch et al. 04]
A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools, Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, Chris Wroe, 2004

[Konstantopoulos 06]
DL Reasoning, Stasinos Konstantopoulos, 2006

[Lee 00]
Semantic Web on XML, Tim Berners-Lee,
http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

[Lee et al. 01]
The Semantic Web, A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities, Scientific American 2001, Tim Berners-Lee, James Hendler and Ora Lassila,
http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&sc=I100322

[Mahmoudi & Müller-Schloer 06]
Towards Ontology-based Embedded Services, G. Mahmoudi, C. Müller-Schloer, 2006

[Noy & McGuinness 01]
Ontology Development 101: A Guide to Creating Your First Ontology, Natalya F. Noy and Deborah L. McGuinness, 2001
http://protege.stanford.edu/publications/ontology_development/ontology101.html

[Scherer & Grinewitschus 02]
Das intelligente Haus: neue Nutzeffekte durch integrierende Vernetzung im Bereich Wohnen und Arbeiten, Dipl.-Ing. Klaus Scherer, Dr.-Ing. Viktor Grinewitschus, 2002

[Schildt & Zenz 05a]
L3 Chat – Eine Beispielapplikation für das L3 Netzwerk, Sebastian Schildt und Gideon Zenz, Universität Lüneburg, 2005

[Schildt & Zenz 05b]
L3 Net Peer to Peer Protokoll, Spezifikation, Sebastian Schildt und Gideon, Universität Lüneburg, 2005

[Stojanovic et al. 04]
Ljiljana Stojanovic, Juergen Schneider, Alexander Maedche, Susanne Libischer, Rudi Studer, Thomas Lumpp, Andreas Abecker, Gerd Breiter, John Dinger: The Role of Ontologies in Autonomic Computing Systems. IBM Systems Journal, Vol. 43 (No. 3). August 2004.

[WCOED 87]
World Commission On Environment and Development. Our Common Future, Oxford University Press, 1987

[Welge 01]
SDL.RT-basierter Entwurf und Implementierung eingebetteter zeit- und sicherheitskritischer Systeme, Ralph Welge, Shaker Verlag, 2001

[Welge 06]
Sensor networking with L3-NET, A SELF-X Middleware based on standard TCP/IP protocols, Embeded World Conference, 2006

8.2 Abbildungsverzeichnis

8.3 Listings

Anhang

JavaDoc der Embedded Ontology Engine

Gideon Zenz

Package

de.unilg.Ontology

de.unilg.Ontology
Class OntologyManager

```
java.lang.Object
   |
   +-de.unilg.Ontology.OntologyManager
```

All Implemented Interfaces:
java.io.Serializable

---

public class **OntologyManager**
extends java.lang.Object
implements java.io.Serializable

Hauptklasse, die die gesamte Ontologie verwaltet und Methoden zur Erzeugung, für Suchanfragen etc. bietet
Author:
Gideon Zenz

Fields

serialVersionUID

```
private static final long serialVersionUID
```

Constant value: **-3695857435443713787**

---

rootClasses

```
private de.unilg.Ontology.SVector rootClasses
```

Enthält alle Rootklassen der Ontologie

---

allClasses

```
private de.unilg.Ontology.SVector allClasses
```

Vector mit allen in der Ontologie enthaltenen Klassen

---

allProperties

```
private de.unilg.Ontology.SVector allProperties
```

Vector mit allen in der Ontologie enthaltenen Properties

---

allIndividuals

```
private de.unilg.Ontology.SVector allIndividuals
```

Vector mit allen in der Ontologie enthaltenen Individuals

---

dummyIndividuals

```
private de.unilg.Ontology.SVector dummyIndividuals
```

(continued on next page)

(continued from last page)

Wenn ein Individual beim deserialisieren referenziert wird, das noch nicht in der Ontologie enthalten ist, wird ein Dummy angelegt und die Ontologie als inkonsistent markiert. Wird im weiteren das Individual doch noch eingelesen, wird der Dummy ersetzt. Dadurch werden zirkuläre Abhänigkeiten unterstützt.

dummyClasses

```
private de.unilg.Ontology.SVector dummyClasses
```

Wenn eine Klasse beim deserialisieren referenziert wird, die noch nicht in der Ontologie enthalten ist, wird ein Dummy angelegt und die Ontologie als inkonsistent markiert. Wird im weiteren die Klasse doch noch eingelesen, wird der Dummy ersetzt. Dadurch werden zirkuläre Abhänigkeiten unterstützt.

dummyProperties

```
private de.unilg.Ontology.SVector dummyProperties
```

Wenn eine Property beim deserialisieren referenziert wird, die noch nicht in der Ontologie enthalten ist, wird ein Dummy angelegt und die Ontologie als inkonsistent markiert. Wird im weiteren die Property doch noch eingelesen, wird der Dummy ersetzt. Dadurch werden zirkuläre Abhänigkeiten unterstützt.

Constructors

OntologyManager

```
public OntologyManager()
```

Methods

getAllClasses

```
public SVector getAllClasses()
```

getAllIndividuals

```
public SVector getAllIndividuals()
```

getAllProperties

```
public SVector getAllProperties()
```

isConsistent

```
public boolean isConsistent()
```

Überprüft, ob die Ontologie konsistent ist, d.h. alle Dummy Individuals/Klassen/Properties aufgelöst wurden. Sollte nach einer vollständigen Deserialisierung aufgerufen werden.

(continued on next page)

(continued from last page)

findIndividual

```
public OWLIndividual findIndividual(java.lang.String Name,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Durchsucht alle Individuals nach dem angegebenen Element. Wird dies nicht gefunden, wird versucht, es über den Retriever zu erhalten

---

findProperty

```
public OWLProperty findProperty(java.lang.String Name,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Durchsucht alle Properties nach dem angegebenen Element. Wird dies nicht gefunden, wird versucht, es über den Retriever zu erhalten

---

findClass

```
private OWLClass findClass(SVector List,
         java.lang.String Name,
         OWLRetriever Retriever)
  throws java.lang.Exception
```

Helpermethode für findClass / findRootClass

---

findClass

```
public OWLClass findClass(java.lang.String Name,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Durchsucht alle Klassen nach dem angegebenen Element. Wird dies nicht gefunden, wird versucht, es über den Retriever zu erhalten

---

findRootClass

```
public OWLClass findRootClass(java.lang.String Name,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Durchsucht die Rootlklassen nach dem angegebenen Element. Wird dies nicht gefunden, wird versucht, es über den Retriever zu erhalten

---

split

```
private SVector split(java.lang.String in)
```

Ersatz für String.split (JDK 1.4)

---

deserializeClass

```
public OWLClass deserializeClass(java.lang.String desc,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Fügt eine serialisierte Klasse in die Ontologie ein. Der Retriever wird benötigt, falls unbekannte Elemente referenziert werden

---

(continued on next page)

(continued from last page)

deserializeProperty

```
public OWLProperty deserializeProperty(java.lang.String desc,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Fügt eine serialisierte Property in die Ontologie ein. Der Retriever wird benötigt, falls unbekannte Elemente referenziert werden

---

deserializeIndividual

```
public OWLIndividual deserializeIndividual(java.lang.String desc,
        OWLRetriever Retriever)
  throws java.lang.Exception
```

Fügt ein serialisierte Individual in die Ontologie ein. Der Retriever wird benötigt, falls unbekannte Elemente referenziert werden

---

findAutonomicUnit

```
public OWLIndividual findAutonomicUnit(java.lang.String uuid)
```

---

removeAutonomicUnit

```
public boolean removeAutonomicUnit(java.lang.String uuid)
```

Entfernt ein AutonomicUnit

---

removeHumanActivity

```
public boolean removeHumanActivity(java.lang.String Name)
```

Entfernt eine HumanActivity.

---

removeIndividual

```
public boolean removeIndividual(OWLIndividual i)
```

Entfernt ein Individual aus der Ontologie. Allerdings wird das Individual nicht richtig entfernt, wenn es in einer Property enthalten ist!

---

createOWLRootClass

```
public OWLClass createOWLRootClass(java.lang.String Name)
```

Erzeugt eine Rootklasse

Returns:

die erzeuget OWLClass oder null

---

createOWLClass

```
public OWLClass createOWLClass(java.lang.String Name,
        SVector Parent)
```

Erzeugt eine Subklasse mit mehreren Eltern

Returns:

(continued on next page)

(continued from last page)

Die erzeugte Klasse oder null

createOWLClass

```
public OWLClass createOWLClass(java.lang.String Name,
        OWLClass Parent)
```

Erzeugt eine Subklasse

Returns:
Die erzeugte Klasse oder null

createOWLClass

```
public OWLClass createOWLClass(java.lang.String Name)
```

Erzeugt eine Klasse

Returns:
Die erzeugte Klasse oder null

createOWLIndividual

```
public OWLIndividual createOWLIndividual(java.lang.String Name)
```

Hilfsklasse für die Deserialisierung. Erzeugt ein Individual

Returns:
Das erzeugte Individual oder null

createOWLIndividual

```
private OWLIndividual createOWLIndividual(java.lang.String Name,
        java.util.HashMap Property)
  throws java.lang.Exception
```

Hilfsklasse für die Deserialisierung. Erzeugt ein Individual mit Properties aber ohne Heritage

Returns:
Das erzeugte Individual oder null

createOWLIndividual

```
public OWLIndividual createOWLIndividual(java.lang.String Name,
        OWLClass heritage)
```

Erzeugt ein Individual einer bestimmten Klasse

Returns:
Das erzeugte Individual oder null

createOWLIndividual

```
public OWLIndividual createOWLIndividual(java.lang.String Name,
        java.util.HashMap Property,
        OWLClass heritage)
  throws java.lang.Exception
```

Erzeugt ein Individual einer bestimmten Klasse mit Properties

(continued on next page)

(continued from last page)

Returns:
Das erzeugte Individual oder null

---

createOWLProperty

```
public OWLProperty createOWLProperty(java.lang.String Name,
        java.lang.Object Range,
        boolean isSingle,
        OWLProperty Parent)
```

Erzeugt eine Subproperty, die eine Range hat und Multiproperty sein kann

Returns:
Die erzeugte Property oder null

---

createOWLProperty

```
public OWLProperty createOWLProperty(java.lang.String Name,
        java.lang.Object Range,
        boolean isSingle)
```

Erzeugt eine Property, die eine Range hat und MultiProperty sein kann

Returns:
Die erzeugte Property oder null

---

createOWLProperty

```
public OWLProperty createOWLProperty(java.lang.String Name,
        java.lang.Object Range)
```

Erzeugt eine Property, die eine Range hat

Returns:
Die erzeugte Property oder null

---

createOWLProperty

```
public OWLProperty createOWLProperty(java.lang.String Name)
```

Erzeugt eine Property

Returns:
Die erzeuget OWLProperty oder null

---

deserializeAll

```
public java.lang.String deserializeAll()
```

de.unilg.Ontology
Class OWLClass

```
java.lang.Object
   |
   +-de.unilg.Ontology.OWLClass
```

All Implemented Interfaces:
java.io.Serializable

---

public class **OWLClass**
extends java.lang.Object
implements java.io.Serializable

OWL Klassen sind die Basis einer ontologischen Bechreibung. OWL Klassen bündeln Individuals durch ihre Properties. Sie können mehrere Subklassen und mehrere Parentklassen besitzen

Author:
Gideon Zenz All Rights Reserved

Fields

serialVersionUID

```
private static final long serialVersionUID
```

Constant value: **-2256632075792815437**

---

name

```
public java.lang.String name
```

Name der Klasse, Kommentar zur Klasse

---

comment

```
public java.lang.String comment
```

Name der Klasse, Kommentar zur Klasse

---

parent

```
public de.unilg.Ontology.SVector parent
```

Klassen, von denen diese abstammt

---

subclass

```
public de.unilg.Ontology.SVector subclass
```

Subklassen der Klasse

(continued from last page)

property

```
public de.unilg.Ontology.SVector property
```

Properties, die diese Klasse besitzt

---

individual

```
public de.unilg.Ontology.SVector individual
```

Vector mit den zur Klasse gehörenden Individuals

Constructors

OWLClass

```
public OWLClass(java.lang.String Name)
```

Konstruktor

OWLClass

```
public OWLClass(java.lang.String Name,
                OWLClass Parent)
```

Konstruktor für Subklassen

OWLClass

```
public OWLClass(java.lang.String Name,
                SVector Parent)
```

Konstruktor für Subklassen, die mehrere Eltern haben

Methods

toString

```
public java.lang.String toString()
```

Überschreibt toString() so daß der Name zurückgeliefert wird

Returns:
Name der Klasse

---

serialize

```
public java.lang.String serialize()
```

Serialisiert die Klasse in String-Form z.B. für die Übertragung über Netz

Returns:
serialisierte Klasse

---

removeIndividual

```
public boolean removeIndividual(OWLIndividual i)
  throws java.lang.Exception
```

(continued from last page)

Entfernt ein Individual aus der Klasse

Returns:
true bei Erfolg

---

inheritProperties

```
public void inheritProperties()
```

Hilfs-Klasse für den Importer. Wird benötigt, wenn die Parent-Klasse erst nach ihren Subklassen in die Ontologie eingefügt wird. Diese Funktion vererbt dann alle Properties an die neuen Subklassen

---

iterateinheritproperties

```
private void iterateinheritproperties(OWLClass parent,
         OWLClass child)
```

Hilfsfunktion für inheritProperties

See Also:
inheritProperties()

---

addSubclass

```
public void addSubclass(OWLClass c)
  throws java.lang.Exception
```

Fügt der Klasse eine Subklasse hinzu. In der neuen Klasse wird diese Klasse als Parent eingetragen, sowie alle Properties vererbt

---

addProperty

```
public void addProperty(OWLProperty p)
  throws java.lang.Exception
```

Fügt eine Property zur Klasse hinzu. Die Domain der Property und deren Super-Properties wird automatisch erweitert

---

addDeepDomain

```
private void addDeepDomain(OWLProperty p)
```

Fügt diese Klasse als Domain auch zu den Eltern-Properties der Property hinzu

See Also:
addProperty(OWLProperty)

---

addIndividual

```
public void addIndividual(OWLIndividual i)
  throws java.lang.Exception
```

Fügt ein Individual zur Klasse hinzu. Das Individual darf nur Properties besitzen, die die Klasse auch besitzt

de.unilg.Ontology
Class OWLIndividual

```
java.lang.Object
   |
   +-de.unilg.Ontology.OWLIndividual
```

All Implemented Interfaces:
java.io.Serializable

---

public class **OWLIndividual**
extends java.lang.Object
implements java.io.Serializable

OWLIndividuals sind Instanzen von einer Klasse. Sie besitzen Properties
Author:
Gideon Zenz

Fields

serialVersionUID

```
private static final long serialVersionUID
```

Constant value: **-4079961157773262900**

---

name

```
public java.lang.String name
```

Name und Kommentar des Individuals

---

comment

```
public java.lang.String comment
```

Name und Kommentar des Individuals

---

heritage

```
public de.unilg.Ontology.OWLClass heritage
```

Klasse, zu dem dieses Individual gehört

---

property

```
private java.util.HashMap property
```

Map mit den Properties der Klasse und ihren Inhalten. Der Key ist eine OWLProperty, die Value ein String, Integer oder ein OWLIndividual, oder ein entsprechender SVector, falls die Property eine MultiProperty ist

Constructors

(continued from last page)

OWLIndividual

```
public OWLIndividual(java.lang.String Name)
```

Defaulkonstruktor

OWLIndividual

```
public OWLIndividual(java.lang.String Name,
                     java.util.HashMap Property)
```

Konstruktor mit Name und Propertyliste

Methods

toString

```
public java.lang.String toString()
```

Überschreibt toString() so daß der Name zurückgeliefert wird

getProperties

```
public java.util.HashMap getProperties()
```

Liefert alle Properties des Individuals

Returns:
HashMap mit allen Properties

serialize

```
public java.lang.String serialize()
```

Serialisiert das Individual in String-Form z.B. für die Übertragung über Netz

findProperty

```
public java.util.Map.Entry findProperty(java.lang.String name)
```

Sucht nach einer Property anhand des Namens

Returns:
Property, oder null wenn nicht gefunden

updateProperty

```
public void updateProperty(OWLProperty p,
         java.lang.Object v)
  throws java.lang.Exception
```

Aktualisiert eine Property mit einem neuen Wert

checkRange

```
private void checkRange(OWLClass p,
         OWLClass v)
  throws java.lang.Exception
```

(continued from last page)

Helferroutine für addProperty. Überprüft rekursiv, ob das Objekt von der gewünschten Klasse ist bzw. von einer abstammt

addProperty

```
public void addProperty(OWLProperty p,
         java.lang.Object v)
  throws java.lang.Exception
```

Füllt eine Property des Individuals mit einem Objekt. Objekte können Primitives oder andere OWLIndividuals sein. In diesem Fall wird sichergestellt, das das Individual in der Range der Property enthalten ist

addProperty

```
public void addProperty(java.util.HashMap Property)
  throws java.lang.Exception
```

Füllt mehrere Properties des Individuals

de.unilg.Ontology
Class OWLProperty

```
java.lang.Object
   |
   +-de.unilg.Ontology.OWLProperty
```

All Implemented Interfaces:
java.io.Serializable

---

public class **OWLProperty**
extends java.lang.Object
implements java.io.Serializable

OWLProperties verbinden Individuals miteinander oder ordnen Individuals Eigenschaften als Strings oder Integers zu
Author:
Gideon Zenz All Rights Reserved

Fields

serialVersionUID

```
private static final long serialVersionUID
```

Constant value: **-2899201893411536954**

---

name

```
public java.lang.String name
```

Name und Kommentar der Property

---

comment

```
public java.lang.String comment
```

Name und Kommentar der Property

---

domain

```
public de.unilg.Ontology.SVector domain
```

Domäne der Property, also Klassen, die diese Property besitzen

---

range

```
public de.unilg.Ontology.SVector range
```

Range der Property. Nur Objekte dieser Art können als Werte aufgenommen werden

---

parent

```
public de.unilg.Ontology.OWLProperty parent
```

(continued from last page)

Eltern-Property falls vorhanden

subproperty

```
public de.unilg.Ontology.SVector subproperty
```

Subproperties der Property

isSingleProperty

```
public boolean isSingleProperty
```

SingleProperties können nur einen Wert aufnehmen, Multiproperties dagegen mehrere

Constructors

OWLProperty

```
public OWLProperty(java.lang.String Name)
```

Konstruktor

OWLProperty

```
public OWLProperty(java.lang.String Name,
                   java.lang.Object Range)
```

Konstruktor, der auch die Range spezifiziert. Range kann ein Objekt oder ein SVector sein. Die Property wird als SingleProperty markiert

OWLProperty

```
public OWLProperty(java.lang.String Name,
                   java.lang.Object Range,
                   boolean IsSingle)
```

Konstruktor, der auch die Range spezifiziert. Range kann ein Objekt oder ein SVector sein. Weiterhin kann angegeben werden, ob die Property eine Single- oder Multiproperty ist, also ob sie eine oder mehrere Werte aufnehmen kann

OWLProperty

```
public OWLProperty(java.lang.String Name,
                   java.lang.Object Range,
                   OWLProperty Parent)
```

Konstruktor für Subproperties, der auch die Range spezifiziert. Range kann ein Objekt oder ein SVector sein. Die Property wird als singleProperty definiert

OWLProperty

```
public OWLProperty(java.lang.String Name,
                   java.lang.Object Range,
                   boolean IsSingle,
                   OWLProperty Parent)
```

Konstruktor für Subproperties, der auch die Range spezifiziert. Range kann ein Objekt oder ein SVector sein. Weiterhin kann angegeben werden, ob die Property eine Single- oder Multiproperty ist, also ob sie eine oder mehrere Werte aufnehmen kann

(continued from last page)

Methods

serialize

```
public java.lang.String serialize()
```

Serialisiert das Individual in String-Form z.B. für die Übertragung über Netz

toString

```
public java.lang.String toString()
```

Überschreibt toString() so daß der Name zurückgeliefert wird

addSubproperty

```
public void addSubproperty(OWLProperty p)
  throws java.lang.Exception
```

Fügt diese Property als Subproperty hinzu. Diese Property wird automatisch als Parent eingetragen

de.unilg.Ontology
Interface OWLRetriever

public interface **OWLRetriever**
extends

Generschies Interface für einen Retriever. Ein Retriever wird bei Anfragen mit an die Ontologie übergeben und benutzt, falls ein benötigtes Objekt nicht gefunden wurde. Die Aufgabe des Retriever ist es, dann in einer anderen Datenquelle nachzusehen. Beispiele sind der Import aus Protege oder der L3Retriever für verteilte Ontologieen

Author:
Gideon Zenz

Methods

getClass

```
public OWLClass getClass(java.lang.String name,
        OntologyManager mgr)
  throws java.lang.Exception
```

Wird aufgerufen, wenn eine Klasse nicht in der Ontologie 'mgr' gefunden werden konnte

Returns:
gesuchte Klasse oder NULL

getIndividual

```
public OWLIndividual getIndividual(java.lang.String name,
        OntologyManager mgr)
  throws java.lang.Exception
```

Wird aufgerufen, wenn ein Individual nicht in der Ontologie 'mgr' gefunden werden konnte

Returns:
gesuchtes Individual oder NULL

getProperty

```
public OWLProperty getProperty(java.lang.String name,
        OntologyManager mgr)
  throws java.lang.Exception
```

Wird aufgerufen, wenn eine Property nicht in der Ontologie 'mgr' gefunden werden konnte

Returns:
gesuchte Property oder NULL

de.unilg.Ontology
Class SVector

```
java.lang.Object
   |
   +-java.util.AbstractCollection
      |
      +-java.util.AbstractList
         |
         +-java.util.Vector
            |
            +-de.unilg.Ontology.SVector
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Cloneable, java.util.Collection, java.util.List, java.io.Serializable, java.lang.Cloneable, java.util.RandomAccess, java.util.List

---

public class **SVector**
extends java.util.Vector
implements java.util.List, java.util.RandomAccess, java.lang.Cloneable, java.io.Serializable, java.util.List, java.util.Collection, java.lang.Cloneable, java.io.Serializable

Erweiterung der Standardklasse Vector um Suchfunktionen
Author:
Gideon Zenz

Fields

serialVersionUID

```
private static final long serialVersionUID
```

Constant value: **-4678878347318576343**

Constructors

SVector

```
public SVector()
```

Überschreibt den Konstruktor, um den Speicherbedarf zu minimieren

Methods

clone

```
public java.lang.Object clone()
```

---

findByField

```
public java.lang.Object findByField(java.lang.Object name,
         java.lang.String Field)
  throws java.lang.Exception
```

(continued from last page)

Durchsucht den Vector nach einem Element, das ein Member "field" besitzt, das das angegebene Objekt enthält. Hierbei wird in jedem Element geprüft, ob die enthaltene Klasse die angegebene Variable besitzt und ihr Inhalt mit dem übergebenen Suchobjekt verglichen

---

find

```
public java.lang.Object find(java.lang.String name)
  throws java.lang.Exception
```

Sucht ein Objekt anhand des Members 'name'

Returns:

Gesuchtes Objekt aus dem Vector

A2. Formale Definition von OWL

Zur Vollständigkeit ist hier eine Übertragung der formalen abstrakten Spezifikation von *OWL* angefügt[15]. Außer dieser existiert noch eine auf der Spezifikation von *RDF* aufsetzende. Da diese jedoch direkt auf der *RDF* Spezifikation aufsetzt, ist sie ohne diese nicht verständlich. Deshalb beschränkt sich die folgende auf die Elemente, die die Sprache ausmachen, ohne sie konkret auf *RDF-XML* herunterzubrechen.

A2.1 OWL Syntax

Im Folgenden wird der Syntax von *OWL* in der erweiterten **Backus-Naur Form** (EBNF) angegeben. Hierbei gilt folgende Konvention:

> Terminale stehen in Anführungszeichen, non-Terminale in **Fettdruck**. Alternativen werden durch ein | angezeigt. Komponenten, die maximal einmal auftreten dürfen, werden in eckigen Klammern […] aufgezählt, solche, die beliebig oft auftreten dürfen, in geschweiften {…}.

A2.1.1 Ontologien

Eine OWL Ontologie besteht aus einer Sequenz von **Anmerkungen**, **Axiomen** und **Fakten**. Der eigentliche Inhalt wird durch **Axiome** und **Fakten** transportiert. Diese beinhalten Informationen über *Klassen*, *Properties* und *Individuals*. Der Name einer Ontologie besteht aus einer **URI** die bezeichnet, wo die Ontologie gefunden werden kann.

ontology ::= '*Ontology(*' [***ontologyID***] { ***directive*** } '*)*'
directive ::= '*Annotation(*' ***ontologyPropertyID ontologyID*** '*)*'
| '*Annotation(*' ***annotationPropertyID URIreference*** '*)*'
| '*Annotation(*' ***annotationPropertyID dataLiteral*** '*)*'
| '*Annotation(*' ***annotationPropertyID individual*** '*)*'
| ***axiom***
| ***fact***

Ontologien bestehen aus Informationen über *Klassen*, *Properties* und *Individuals*, die alle über eine **URI** referenziert werden.

datatypeID ::= ***URIreference***
classID ::= ***URIreference***
individualID ::= ***URIreference***
ontologyID ::= ***URIreference***
datavaluedPropertyID ::= ***URIreference***
individualvaluedPropertyID ::= ***URIreference***
annotationPropertyID ::= ***URIreference***
ontologyPropertyID ::= ***URIreference***

Dabei ist ein *Datatype* definiert als die Menge der Werte, aus denen sein Wertebereich bestehet.

Klassen bestehen aus einer Menge von *Individuals.*

[15] Siehe http://www.w3.org/TR/2004/REC-owl-semantics-20040210/

Properties verbinden *Individuals* mit anderen Informationen. Sie bestehen aus 4 disjunkten Gruppen:

- Daten *Properties*
  Zuordnung *Individual* → Datenwert

- Individual *Properties*
  Zuordnung *Individual* → *Individual*

- Anmerkungs *Properties*
  Dies sind Anmerkungen zu *Individuals*, *Klassen*, *Propertiesy* sowie Ontologien

- Ontology Properties
  Zuordnung Ontologie → Ontologie, z.B. zum Import

Es gibt zwei *OWL* inhärente *Klassen*, **owl:Thing**, die *Klasse* aller *Individuals*, und **owl:Nothing**, die leere *Klasse.*

Weiterhin unterstützt *OWL* die meisten **XML Schema** Datentypen[16].

Die Definition von Anmerkungen ist:

annotation ::= '*annotation(*' ***annotationPropertyID URIreference*** '*)*'
| '*annotation(*' ***annotationPropertyID dataLiteral*** '*)*'
| '*annotation(*' ***annotationPropertyID individual*** '*)*'

A2.1.2 Fakten

Fakten unterteilen sich in zwei Arten. Die erste akkumuliert Informationen zu einem spezifischen *Individual*, nämlich zu welcher *Klasse* es gehört und welche *Properties/Values* zu ihm gehören.

fact ::= ***individual***
individual ::= '*Individual(*' [***individualID***] { ***annotation*** }
{ '*type(*' ***type*** '*)*' } { ***value*** } '*)*'

value ::= '*value(*' ***individualvaluedPropertyID individualID*** '*)*'
| '*value(*' ***individualvaluedPropertyID individual*** '*)*'
| '*value(*' ***datavaluedPropertyID dataLiteral*** '*)*'

Fakten sind in *OWL Lite* und *OWL DL* bis auf den *Typ* identisch. Bei *OWL Lite* können *Typen* nur *Klassen* oder *Restriktionen* sein:

type ::= ***classID***
| ***restriction***

Bei *OWL DL* können sie allgemeine *Deskriptionen* sein, was die Definition von *OWL Lite* einschließt, aber weiter gefasst ist:

type ::= ***description***

Die oben genutzten *dataLiteras* können entweder *typisiert* oder *einfach* sein. *Einfache* bestehen aus einer *Unicode*-Zeichenkette in *Normalform C* mit einem optionalen

[16] http://www.w3.org/TR/2004/REC-owl-semantics-20040210/#ref-xmls-datatypes

Sprach-*Tag*. *Typisierte* Literale sind die lexikalische Repräsentation einer *URI*. Beides ist genauso wie in *RDF* definiert:

dataLiteral ::= ***typedLiteral*** | ***plainLiteral***
typedLiteral ::= ***lexicalForm*^^*URIreference***
plainLiteral ::= ***lexicalForm*** | ***lexicalForm*@*languageTag***
lexicalForm ::= *wie in RDF, eine Unicode-Zeichenkette in Normalform C*
languageTag ::= *wie in RDF, ein XML Language Tag*

Die zweite Art von *Fakten* wird benutzt, um *Individuals* als gleich oder paarweise disjunkt zu bezeichnen:

fact ::= '*SameIndividual(*' ***individualID individualID*** {***individualID***} '*)*'
| '*DifferentIndividuals(*' ***individualID individualID*** {***individualID***}'*)*'

A2.1.3 Axiome

Axiome assoziieren *Klassen* und *Properties* mit ihrer Spezifikation. Man könnte auch *Definition* anstatt *Axiom* sagen, allerdings unterscheiden sich *Axiome* vom allgemeinen Verständnis einer Definition, von daher wurde ein neutralerer Name gewählt.

Bei Axiomen unterscheiden sich *OWL DL* und *OWL Lite* am deutlichsten. *OWL Lite* stellt hierbei ein Spezialfall von OWL DL dar.

A2.1.3.1 OWL Lite Klassenaxiome

Hiermit können *Klassen* als exakt gleich (*modality complete*) oder als *Subklasse (modality partial)* einer Vereinigung einer Menge von *Superklassen* oder *OWL Lite Restrictions* spezifiziert werden.

axiom ::= '*Class(*' ***classID*** ['*Deprecated*'] ***modality***
{ ***annotation*** } { ***super*** } '*)*'

modality ::= '*complete*' | '*partial*'
super ::= ***classID*** | ***restriction***

Weiterhin lassen sich *Klassen* als äquivalent spezifizieren:

axiom ::= '*EquivalentClasses(*' ***classID classID*** { ***classID*** } '*)*'

Schließlich gibt es noch *Datentyp Axiome*. Diese verbinden den *Datentypen* mit einer *DatatypeID* und können *Anmerkung* speichern.

axiom ::= '*Datatype(*' ***datatypeID*** ['*Deprecated*'] { ***annotation*** } *)*'

A2.1.3.2 OWL Lite Restriktionen

Mit *Restriktionen* können lokale Beschränkungen von *Properties* einer *Klasse* definiert werden.

Im einzelnen sind möglich:

- **allValuesFrom**
  alle Werte der beschränkten *Property* dieser *Klasse* müssen zu dem spezifizierten *Datentyp* bzw. der spezifizierten *Klasse* gehören.

- **someValuesFrom**
  mindestens ein Wert der beschränkten *Property* dieser *Klasse* muss zu dem spezifizierten *Datentyp* bzw. der spezifizierten *Klasse* gehören.

- **Cardinality**
  spezifiziert die Anzahl der verschiedenartigen Werte, die die beschränkten *Property* dieser *Klasse* besitzen darf.

Bei *OWL Lite* kann die Cardinality jedoch nur 0 oder 1 sein.

restriction ::= '*restriction(*' ***datavaluedPropertyID*** ***dataRestrictionComponent*** '*)*'
| '*restriction(*' ***individualvaluedPropertyID*** ***individualRestrictionComponent*** '*)*'

dataRestrictionComponent
::= '*allValuesFrom(*' ***dataRange*** '*)*'
| '*someValuesFrom(*' ***dataRange*** '*)*'
| ***cardinality***

individualRestrictionComponent
::= '*allValuesFrom(*' ***classID*** '*)*'
| '*someValuesFrom(*' ***classID*** '*)*'
| ***cardinality***

cardinality ::= '*minCardinality(0)*' | '*minCardinality(1)*'
| '*maxCardinality(0)*' | '*maxCardinality(1)*'
| '*cardinality(0)*' | '*cardinality(1)*'

A2.1.3.3 OWL Lite Property-Axiome

Es gibt zwei Arten von *Properties*, nämlich *Daten Properties*, die *Individuals* mit *Datenwerten* wie Integer etc. assoziieren und *Object Properties*, die *Individuals* mit anderen *Individuals* assoziieren. Beide Arten können *Super-Properties* haben und so eine Hierarchie konstruieren. Beide können auch Werte- und Bildbereiche (*Domains* und *Ranges*) haben.

Wie in *RDFS* spezifiziert eine *Domain*, welche *Individuals* potenzielle Subjekte von *Statements* sind, die diese *Property* als Prädikat besitzen. In *OWL Lite* sind *Domains* immer *Klassen*. Eine *Property* kann auch mehrere *Domains* haben.

Die Range spezifiziert, auf welche *Individuals* oder Datenwerte die *Property* verweisen kann. Eine *Property* kann mehrere *Ranges* haben.

Daten-Properties können als *Functional* deklariert werden, d.h. sie assoziieren ein *Individual* nur zu maximal einem *Datenwert*.

Object-Properties können zusätzlich auch als *Inverses* einer anderen *Object-Property* deklariert werden sowie als *symetrisch* bzw. *transitiv* zu einer.

axiom ::= '*DatatypeProperty(*' ***datavaluedPropertyID*** ['*Deprecated*']
{ ***annotation*** } { '*super(*' ***datavaluedPropertyID*** '*)*' }
['*Functional*'] { '*domain(*' ***classID'*** '*)*' }
{ '*range(*' ***dataRange*** '*)*' } '*)*'

| '*ObjectProperty(*' ***individualvaluedPropertyID*** ['*Deprecated*']
{ ***annotation*** } {'*super(*' ***individualvaluedPropertyID*** '*)*' }
['*inverseOf(*' ***individualvaluedPropertyID*** '*)*']
['*Symmetric*'] ['*Functional*' | '*InverseFunctional*' |
'*Functional*' '*InverseFunctional*' | '*Transitive*']
{ '*domain(*' ***classID*** '*)*' } { '*range(*' ***classID*** '*)*' } '*)*'

| '*AnnotationProperty(*' ***annotationPropertyID*** { ***annotation*** } '*)*'

| '*OntologyProperty(*' ***ontologyPropertyID*** { ***annotation*** } '*)*'

dataRange ::= ***datatypeID*** | '*rdfs:Literal*'

Mit dem folgenden Axiom können verschiedene *Properties* als *äquivalent* bzw. als S*ub-Property* einer anderen deklariert werden.

axiom ::= '*EquivalentProperties(*' ***datavaluedPropertyID***
datavaluedPropertyID { ***datavaluedPropertyID*** } '*)*'

| '*SubPropertyOf(*' ***datavaluedPropertyID***
datavaluedPropertyID '*)*'

| '*EquivalentProperties(*' ***individualvaluedPropertyID***
individualvaluedPropertyID { ***individualvaluedPropertyID*** '*)*'

| '*SubPropertyOf(*' ***individualvaluedPropertyID***
individualvaluedPropertyID '*)*'

A2.1.3.4 OWL DL Klassenaxiome

Die *OWL DL* Syntax ist eine generalisierte Version der *OWL Lite* Syntax. *Superklassen*, erweiterte *Restriktionen* und boolsche Kombinationen von ihnen sind erlaubt. Diese Konstrukte werden *descriptions* genannt. Die *modality* ist genauso wie in *OWL Lite* definiert.

axiom ::= '*Class(*' ***classID*** ['*Deprecated*'] ***modality*** { ***annotation*** }
{ ***description*** } '*)*'

modality ::= '*complete*' | '*partial*'

Weiterhin ist es möglich, dass eine Klasse aus einer genau definierten Menge von *Individuals* besteht:

axiom ::= '*EnumeratedClass(*' ***classID*** ['*Deprecated*'] { ***annotation*** }
{ ***individualID*** } '*)*'

Abschließend können Mengen von *descriptions* als paarweise disjunkt spezifiziert werden. Es kann angegeben werden, dass sie die gleichen *Instanzen* besitzen oder dass eine *description* Subklasse einer anderen ist.

axiom ::= '*DisjointClasses(*' ***description description*** { ***description*** }'*)*'
| '*EquivalentClasses(*' ***description*** { ***description*** } '*)*'
| '*SubClassOf(*' ***description description*** '*)*'

Datentyp Axiome sind genauso wie in *OWL Lite* definiert:

axiom ::= '*Datatype(*' ***datatypeID*** ['*Deprecated*'] { ***annotation*** } *)*'

A2.1.3.5 OWL DL Descriptions

Descriptions beschreiben *Klassen* und *Restriktionen*. Sie können durch eine boolschen Kombination anderer *Descriptions* oder eine Menge von *Individuals* definiert werden.

description ::= ***classID***
| ***restriction***
| '*unionOf(*' { ***description*** } *)*'
| '*intersectionOf(*' { ***description*** } *)*'
| '*complementOf(*' ***description*** *)*'
| '*oneOf(*' { ***individualID*** } *)*'

A2.1.3.6 OWL DL Restrictions

OWL DL Restriktionen generalisieren wiederum die *Restriktionen* von *OWL Lite*, indem sie mit *descriptions* anstatt *Klassen* arbeiten und nicht nur *Datentypen*, sondern auch Mengen von *Datenwerten* zulassen. Die Kombination von *Datentypen* und Mengen von *Datenwerten* heißt *Range*. Weiterhin können auch Kardinalitäten größer 1 benutzt werden.

restriction ::= '*restriction(*' ***datavaluedPropertyID***
dataRestrictionComponent { ***dataRestrictionComponent*** }*')*'

| '*restriction(*' ***individualvaluedPropertyID***
individualRestrictionComponent
{ ***individualRestrictionComponent*** } *)*'

dataRestrictionComponent ::= '*allValuesFrom(*' ***dataRange*** *)*'
| '*someValuesFrom(*' ***dataRange*** *)*'
| '*value(*' ***dataLiteral*** *)*'
| ***cardinality***

individualRestrictionComponent ::= '*allValuesFrom(*' ***description*** *)*'
| '*someValuesFrom(*' ***description*** *)*'
| '*value(*' ***individualID*** *)*'
| ***cardinality***

cardinality ::= '*minCardinality(*' ***non-negative-integer*** *)*'
| '*maxCardinality(*' ***non-negative-integer*** *)*'
| '*cardinality(*' ***non-negative-integer*** *)*'

Eine *Range* ist entweder ein *Datentyp* oder eine Menge von *Datenwerten*:

dataRange ::= ***datatypeID*** | '*rdfs:Literal*'
|'*oneOf(*' { ***dataLiteral*** } *)*'

A2.1.3.7 OWL DL Property Axiome

Auch die *Properties* generalisieren die von *OWL Lite,* indem sie *descriptions* anstatt von Klassen sowie *Ranges* erlauben.

axiom ::= '*DatatypeProperty(*' ***datavaluedPropertyID*** ['*Deprecated*']
{ ***annotation*** } { '*super(*' ***datavaluedPropertyID*** '*)*'}
['*Functional*'] { '*domain(*' ***description*** '*)*' }
{ '*range(*' ***dataRange*** '*)*' } '*)*'

| '*ObjectProperty(*' ***individualvaluedPropertyID*** ['*Deprecated*']
{ ***annotation*** } { '*super(*' ***individualvaluedPropertyID*** '*)*'}
['*inverseOf(*' ***individualvaluedPropertyID*** '*)*']
['*Symmetric*']['*Functional*' | '*InverseFunctional*' |
'*Functional*' '*InverseFunctional*' | '*Transitive*']
{ '*domain(*' ***description*** '*)*' }{ '*range(*' ***description*** '*)*' }
'*)*'

| '*AnnotationProperty(*' ***annotationPropertyID*** { ***annotation*** }'*)*'

| '*OntologyProperty(*' ***ontologyPropertyID*** { ***annotation*** } '*)*'

Wie auch in *OWL Lite* gibt es ein *Axiom*, das erlaubt *Properties* als äquivalent oder als *Sub-property* zu definieren:

axiom ::= '*EquivalentProperties(*' ***datavaluedPropertyID***
datavaluedPropertyID { ***datavaluedPropertyID*** } '*)*'

| '*SubPropertyOf(*' ***datavaluedPropertyID***
datavaluedPropertyID '*)*'

| '*EquivalentProperties(*' ***individualvaluedPropertyID***
individualvaluedPropertyID { ***individualvaluedPropertyID*** }'*)*'

| '*SubPropertyOf(*' ***individualvaluedPropertyID***
individualvaluedPropertyID '*)*'

A2.2 OWL Semantik

Zunächst soll das Vokabular definiert werden. In einer *OWL* Ontologie muss das Vokabular alle **URI** Referenzen und *Literale* enthalten sowie Ontologien, die importiert werden.

A2.2.1 Vokabular und Interpretationen

Definition: Ein ***OWL Vokabular*** *V* besteht aus der Menge der Literale V_L und sieben Mengen von URI Referenzen:

- V_C : Namen der Klassen inklusive **owl:Thing** und **owl:Nothing**
- V_D : Namen der *Datentypen*, inkl. der *OWL*-internen und **rdfs:Literal**
- V_I : Namen der *Individuals*
- V_{DP} : Namen der *Datentyp-Properties*
- V_{IP} : Namen der *Object-Properties*
- V_{AP} : Namen der *Anmerkungen*
- V_O : Namen der Ontologien

Im Folgenden ist V_{OP} die Menge der **URI** Referenzen zu *OWL*-internen *Properties*.

Weiterhin gilt:

- $V_C \cap V_D = \varnothing$
- $\bigcap\{V_{DP}, V_{IP}, V_{AP}, V_{OP}\} = \varnothing$

Definition: Wie bei *RDF* wird ein ***Datentyp*** *d* durch seinen **lexikalischen Raum** *L(d)*, einen **Werteraum** *V(d)* und eine **Abbildung** *L2V(d)* definiert, wobei *L(d)* eine Menge von **Unicode**-Zeichenfolgen ist. *L2V* bildet den **lexikalischen Raum** vollständig auf den **Werteraum** ab.

Definition: Eine ***Datentypmap*** *D* ist eine (nicht vollständige) Abbildung von **URI** Referenzen auf **Datentypen**. Sie bildet **xsd:string** und **xsd:integer** auf die entsprechenden *XML Schema* **Datentypen** ab.

Definition: Sei *D* eine **Datentypmap**. Eine ***abstrakte OWL Interpretation*** *I* in Bezug auf *D* ist ein Tupel *I*=<*R*, *EC*, *ER*, *L*, *S*, *LV*> für das gilt:

- *R* sind die Ressourcen von *I*, $R \neq \varnothing$
- LV sind die Literalwerte von I, wobei $LV \subseteq R$, nämlich die der **Unicode** Zeichenfolgen, der **Sprach-Tags** und der **Werteräume** aller **Datentypen** in D
- $EC : V_C \rightarrow \mathcal{P}(\mathrm{O})$, die Abbildung aller **Klassen** auf die Potenzmenge der Ontologien
- $EC : V_D \rightarrow \mathcal{P}(\mathrm{LV})$, die Abbildung aller **Datentypen** auf die Potenzmenge der Literalwerte
- $ER : V_{DP} \rightarrow \mathcal{P}(\mathrm{O} \times \mathrm{LV})$, die Abbildung aller **Datentyp-Properties** auf die Potenzmenge der **Literalwerte** und **Ontologien**
- $ER : V_{IP} \rightarrow \mathcal{P}(\mathrm{O} \times \mathrm{O})$, die Abbildung aller **Objekt-Properties** auf die Potenzmenge der **Ontologien**
- $ER : V_{AP} \cup \{rdf : type\} \rightarrow \mathcal{P}(\mathrm{R} \times \mathrm{R})$, die Abbildung aller **Anmerkungs-Properties** auf die Potenzmenge der **Ressourcen**
- $ER : V_{OP} \rightarrow \mathcal{P}(\mathrm{R} \times \mathrm{R})$, die Abbildung aller *OWL-internen* **Properties** auf die Potenzmenge der **Ressourcen**
- $L : TL \rightarrow LV$, wobei TL die **typisierten Literale** aus V_L sind
- $S : V_I \cup V_C \cup V_D \cup V_{DP} \cup V_{IP} \cup V_{AP} \cup V_O \cup \{$ **owl:Ontology, OWL:DeprecatedClass, owl:DeprecatedProperty** $\} \rightarrow R$, die Abbildung aller **URI** Referenzen auf ihre **Ressourcen**
- $S(V_I) \subseteq O$, die **Ressourcen** der *Individuals* müssen also Teilmenge der Ontologie sein

Das bedeutet:

- *EC* definiert die Bedeutung der **URI** Referenzen der *Klassen* und *Datentypen*,
- *ER* die der *Properties*,
- *L* die der *typisierten Literale*,
- *S* die zur Bezeichnung von *Individuals* genutzten sowie der *Anmerkungen.*

A2.2.2 Interpretation von eingebetteten Konstrukten

EC wird durch die folgenden Konstrukte syntaktisch erweitert und interpretiert somit die oben definierte Syntax:

EC Extension Table	
Abstract Syntax	**Interpretation (Wert von EC)**
*complementOf(*c*)*	O - EC(c)
unionOf(c_1 … c_n*)*	$EC(c_1) \cup \ldots \cup EC(c_n)$
intersectionOf(c_1 … c_n*)*	$EC(c_1) \cap \ldots \cap EC(c_n)$
oneOf(i_1 … i_n*)*, für Individuals i_j	$\{S(i_1), \ldots, S(i_n)\}$
oneOf(v_1 … v_n*)*, für Literale v_j	$\{S(v_1), \ldots, S(v_n)\}$
*restriction(*p x_1 … x_n*)*, n > 1	EC(*restriction(*p x_1*)*) ∩… ∩ EC(*restriction(*p x_n*)*)
*restriction(*p *allValuesFrom(*r*))*	{x ∈ O | <x,y> ∈ ER(p) → y ∈ EC(r)}
*restriction(*p *someValuesFrom(*e*))*	{x ∈ O | ∃ <x,y> ∈ ER(p) ∧ y ∈ EC(e)}
*restriction(*p *value(*i*))*, für Individual i	{x ∈ O | <x,S(i)> ∈ ER(p)}
*restriction(*p *value(*v*))*, für Literal v	{x ∈ O | <x,S(v)> ∈ ER(p)}
*restriction(*p *minCardinality(*n*))*	{x ∈ O | card({y ∈ O∪LV : <x,y> ∈ ER(p)}) ≥ n}
*restriction(*p *maxCardinality(*n*))*	{x ∈ O | card({y ∈ O∪LV : <x,y> ∈ ER(p)}) ≤ n}
*restriction(*p *cardinality(*n*))*	{x ∈ O | card({y ∈ O∪LV : <x,y> ∈ ER(p)}) = n}
Individual(annotation(p_1 o_1*)* … *annotation(*p_k o_k*)* *type(*c_1*)* … *type(*c_m*)* pv_1 … pv_n*)*	EC(*annotation(*p_1 o_1*)*) ∩ … EC(*annotation(*p_k o_k*)*) ∩ $EC(c_1) \cap \ldots \cap EC(c_m) \cap EC(pv_1) \cap \ldots$ ∩ EC(pv_n)
*Individual(*i *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* *type(*c_1*)* … *type(*c_m*)* pv_1 … pv_n*)*	{S(i)} ∩ EC(*annotation(*p_1 o_1*)*) ∩ … EC(*annotation(*p_k o_k*)*) $\cap EC(c_1) \cap \ldots \cap EC(c_m) \cap EC(pv_1) \cap \ldots$ ∩ EC(pv_n)
*value(*p *Individual(…))*	{x ∈ O | ∃ y∈EC(*Individual(…)*) : <x,y> ∈ ER(p)}
*value(*p id*)* for id an individual ID	{x ∈ O | <x,S(id)> ∈ ER(p) }
*value(*p v*)* for v a literal	{x ∈ O | <x,S(v)> ∈ ER(p) }
*annotation(*p o*)*, o eine **URI** Referenz	{x ∈ R | <x,S(o)> ∈ ER(p) }
*annotation(*p *Individual(…))*	{x ∈ R | ∃ y ∈ EC(*Individual(…)*) : <x,y> ∈ ER(p) }

A2.2.3 Interpretation von Axiomen und Fakten

Weiterhin erfüllt eine **abstrakte OWL Interpretation** I die folgenden *Axiome* und *Fakten*:

Interpretation von Axiomen und Fakten	
Direktive	**Anforderung an die Interpretation**
*Class(*c [*Deprecated*] *complete* *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* $descr_1$ … $descr_n$*)*	[<S(c),S(*owl:DeprecatedClass*)> ∈ ER(*rdf:type*)] S(c) ∈ EC(*annotation(*p_1 o_1*)*) … S(c) ∈ EC(*annotation(*p_k o_k*)*) EC(c) = EC($descr_1$) ∩…∩ EC($descr_n$)
*Class(*c [*Deprecated*] *partial* *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* $descr_1$ … $descr_n$*)*	[<S(c),S(*owl:DeprecatedClass*)> ∈ ER(*rdf:type*)] S(c) ∈ EC(*annotation(*p_1 o_1*)*) … S(c) ∈ EC(*annotation(*p_k o_k*)*) EC(c) ⊆ EC($descr_1$) ∩…∩ EC($descr_n$)
*EnumeratedClass(*c [*Deprecated*] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* i_1 … i_n*)*	[<S(c),S(*owl:DeprecatedClass*)> ∈ ER(*rdf:type*)] S(c) ∈ EC(*annotation(*p_1 o_1*)*) … S(c) ∈ EC(*annotation(*p_k o_k*)*) EC(c) = { S(i_1), …, S(i_n) }
*Datatype(*c [*Deprecated*] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*))*	[<S(c),S(*owl:DeprecatedClass*)> ∈ ER(*rdf:type*)] S(c) ∈ EC(*annotation(*p_1 o_1*)*) … S(c) ∈ EC(*annotation(*p_k o_k*)*) EC(c) ⊆ LV
DisjointClasses(d_1 … d_n*)*	EC(d_i) ∩ EC(d_j) = { } , 1 ≤ i < j ≤ n
EquivalentClasses(d_1 … d_n*)*	EC(d_i) = EC(d_j), 1 ≤ i < j ≤ n
SubClassOf(d_1 d_2*)*	EC(d_1) ⊆ EC(d_2)
*DatatypeProperty(*p [*Deprecated*] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* *super(*s_1*)* … *super(*s_n*)* *domain(*d_1*)* … *domain(*d_n*)* *range(*r_1*)* … *range(*r_n*)* [*Functional*])	[<S(c),S(*owl:DeprecatedProperty*)> ∈ ER(*rdf:type*)] S(p) ∈ EC(*annotation(*p_1 o_1*)*) … S(p) ∈ EC(*annotation(*p_k o_k*)*) ER(p) ⊆ O×LV ∩ ER(s_1) ∩…∩ ER(s_n) ∩ EC(d_1)×LV ∩…∩ EC(d_n)×LV ∩ O×EC(r_1) ∩…∩ O×EC(r_n) [ER(p) ist *Functional*]
*ObjectProperty(*p [*Deprecated*] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* *super(*s_1*)* … *super(*s_n*)* *domain(*d_1*)* … *domain(*d_n) *range(*r_1*)* … *range(*r_n*)* [*inverse(*i)] [*Symmetric*] [*Functional*] [*InverseFunctional*]	[<S(c),S(*owl:DeprecatedProperty*)> ∈ ER(*rdf:type*)] S(p) ∈ EC(*annotation(*p_1 o_1*)*) … S(p) ∈ EC(*annotation(*p_k o_k*)*) ER(p) ⊆ O×O ∩ ER(s_1) ∩…∩ ER(s_n) ∩ EC(d_1)×O ∩…∩ EC(d_n)×O ∩ O×EC(r_1) ∩…∩ O×EC(r_n)

Interpretation von Axiomen und Fakten	
Direktive	**Anforderung an die Interpretation**
[*Transitive*])	[ER(p) ist Inverses von ER(i)] [ER(p) ist symmetrisch] [ER(p) ist functional] [ER(p) ist invers Functional] [ER(p) ist transitiv]
AnnotationProperty(p *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*))*	S(p) ∈ EC(*annotation(*p_1 o_1*)*) … S(p) ∈ EC(*annotation(*p_k o_k*)*)
OntologyProperty(p *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*))*	S(p) ∈ EC(*annotation(*p_1 o_1*)*) … S(p) ∈ EC(*annotation(*p_k o_k*)*)
EquivalentProperties(p_1 … p_n*)*	$ER(p_i) = ER(p_j)$, $1 \le i < j \le n$
SubPropertyOf(p_1 p_2*)*	$ER(p_1) \subseteq ER(p_2)$
SameIndividual(i_1 … i_n*)*	$S(i_j) = S(i_k)$, $1 \le j < k \le n$
DifferentIndividuals(i_1 … i_n*)*	$S(i_j) \neq S(i_k)$ for $1 \le j < k \le n$
Individual([i] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* *type(*c_1*)* … *type(*c_m*)* pv_1 … pv_n*)*	EC(*Individual(*[i] *annotation(*p_1 o_1*)* … *annotation(*p_k o_k*)* *type(*c_1*)* … *type(*c_m*)* pv_1 … pv_n*))* ist nicht leer

A2.2.4 Interpretation von Ontologien

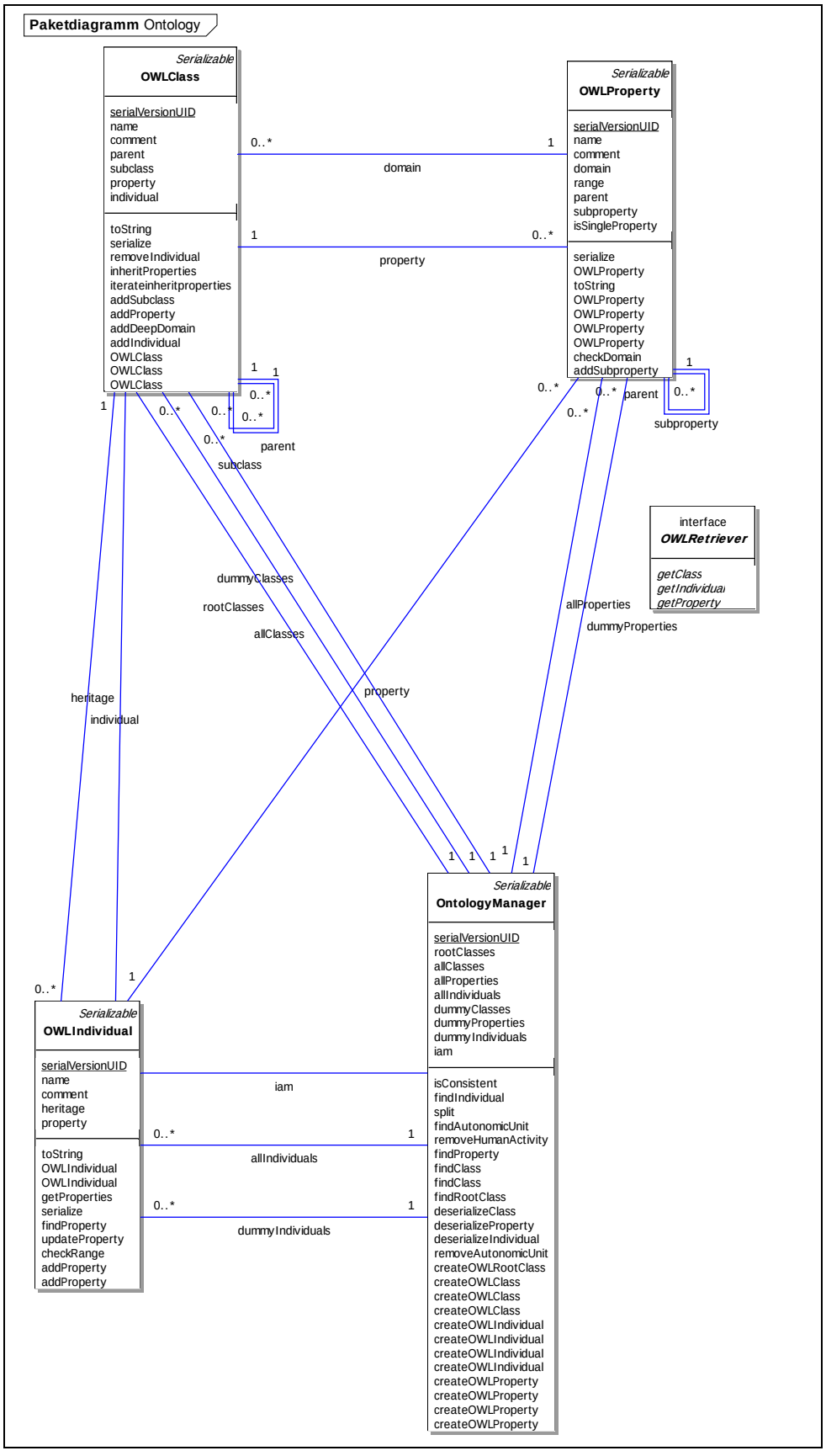
Definition: Sei *D* eine **Datentypmap**. Eine ***abstrakte OWL Interpretation*** *I* bezüglich D mit dem Vokabular $V_L, V_C, V_D, V_I, V_{DP}, V_{IP}, V_{AP}, V_O$ **erfüllt** eine *OWL* Ontologie O genau dann, wenn:

1. Jede in *O* als *Klasse*, *Datentyp*, *Individual, Dataproperty, Object-Property, Anmerkungs-Properyt*, Anmerkung oder Ontologie benutzte **URI** Referenz ist in $V_C, V_D, V_I, V_{DP}, V_{IP}, V_{AP}$ bzw. V_O enthalten
2. Jedes Literal aus O ist in V_L enthalten
3. *I* erfüllt jede Direktive von *O*, außer für Anmerkungen
4. $\exists o \in R, < o, S(\text{owl:Ontology}) > \in ER(\text{rdf:type})$
   $$\Rightarrow \begin{cases} \forall\ \text{Annotation}(p \quad v) :< o, S(v) > \in ER(p) \\ \text{O hat Namen n} \Rightarrow S(n) = o \end{cases}$$
5. *I* erfüllt ebenso jede durch *O* importierte Ontologie

Definition: Eine Menge abstrakter *OWL* Ontologien, *Axiomen* und *Fakten* ist **konsistent** bezüglich einer **Datatypmap** *D*

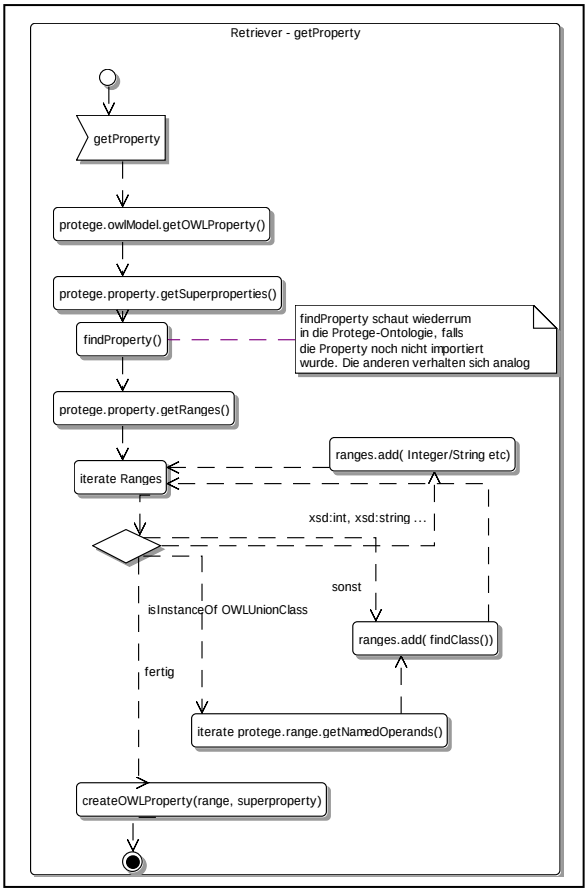
⇔ Es gibt eine **Interpretation** *I* bezüglich *D*, die jede *Ontologie*, jedes *Axiom* und jeden *Fakt* dieser Menge erfüllt.

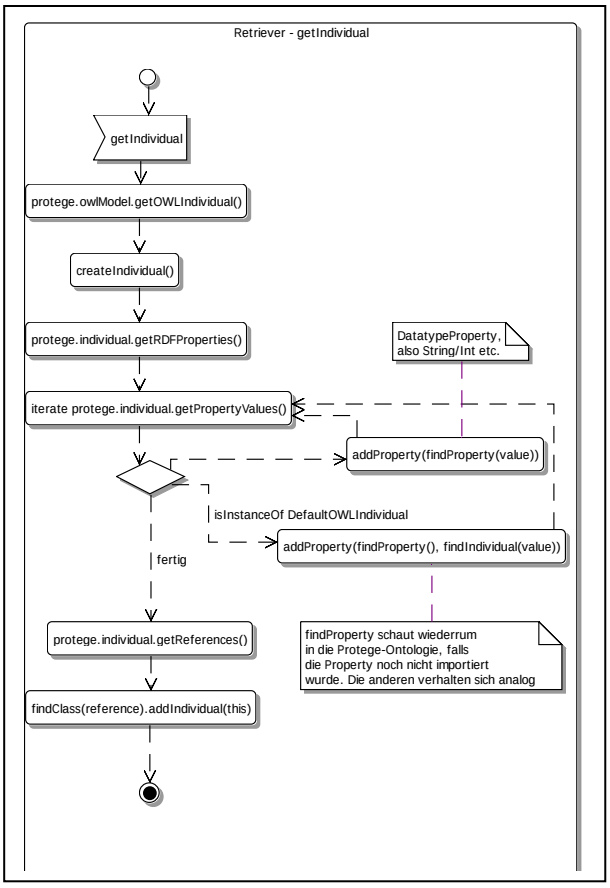
A3. Vollständiges Klassendiagramm der Ontology Engine

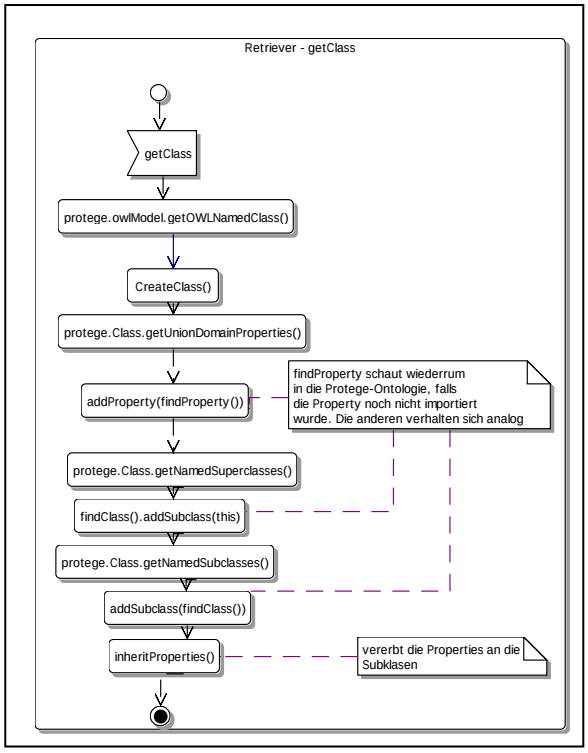


A4. Aktivitätsdiagramme des Importers

Die folgenden Diagramme zeigen die Aktivitäten der drei *Klassen* des **OWL Retrievers** des **Importers** von *OWL-XML* Ontologien. Die Aufgabe jeder dieser *Klassen* ist es, alle *Klasse/Property/Individual* vollständig einzulesen und mittels der **Ontology Engine** nachzubilden. Insbesondere werden dabei weitere fehlende Elemente der Ontologie, also *Subklassen, Properties*, sonstige referenzierte Elemente etc. wiederum erzeugt. Dies geschieht dadurch, dass auch hier die **Ontology Engine** den zuständigen *Retriever* aufruft, wenn ein unbekanntes Element angefragt wird.



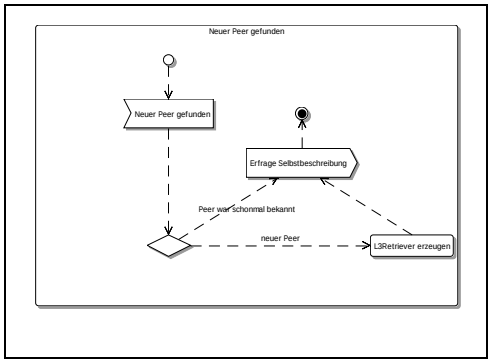




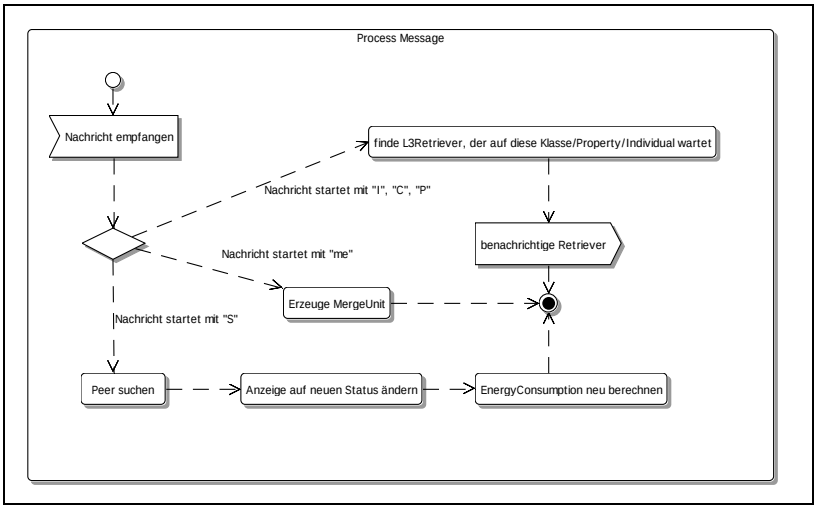
A5. Aktivitätsdiagramme Organic Room Software

In der OrganicRoom Software laufen zwei Pfade, einmal der von den Methoden **ProcessMessage** bzw. **discoveryPeerInfoChanged**, die die empfangene L3 Nachrichten verarbeiten und von **ActionPerformed**, die Benutzerinteraktionen verarbeitet.

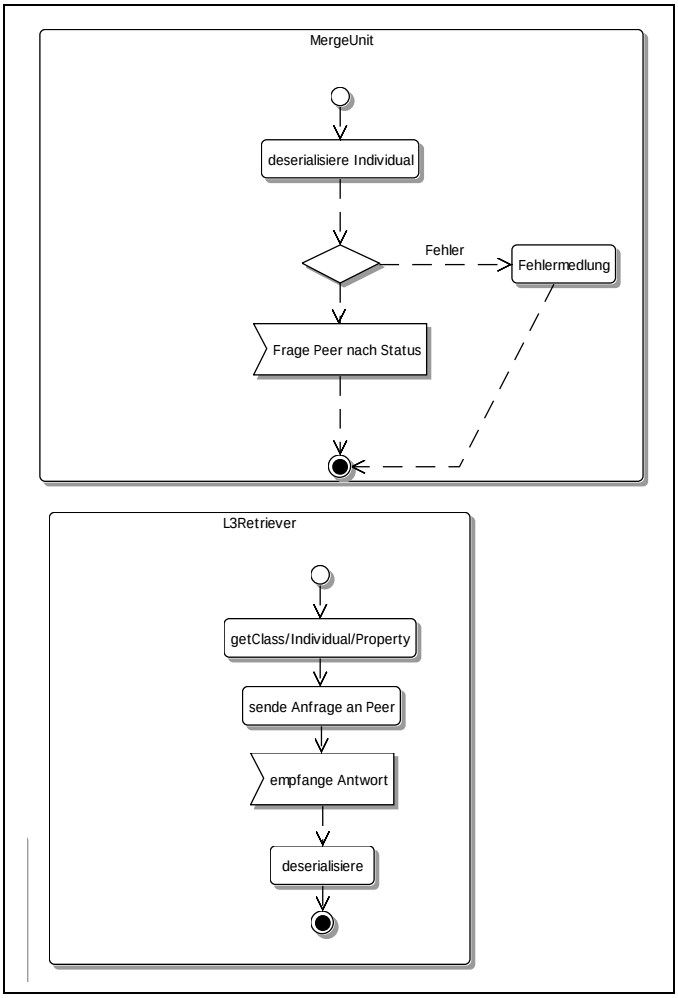
Wird ein neuer Peer gefunden, so wird gegebenenfalls ein neuer **Retriever** instanziert und der Peer danach nach seiner Selbstbeschreibung gefragt. Der **Retriever** ist dafür verantwortlich, unbekannte ontologische Informationen von diesem Peer zu erfragen.



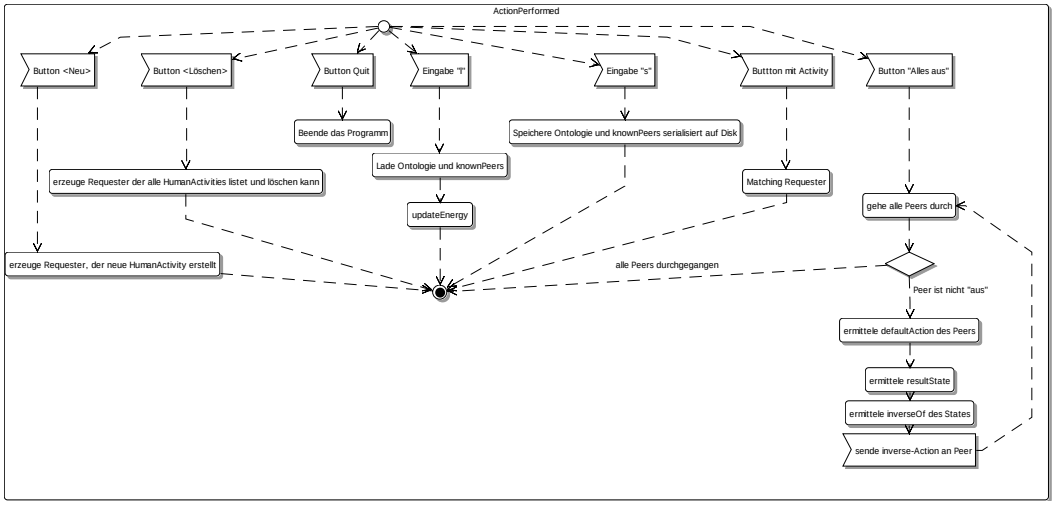
Die Nachrichten von **ProcessMessage** betreffen empfangene ontologische Daten oder den Status von Geräten. Im ersten Fall wird **MergeUnit** instanziert, falls Daten über ein neues Gerät empfangen wurden, ansonsten werden ontologische Teildaten (wie *Instanzen, Klassen, Properties*) an den dann schon existierenden Retriever weitergeleitet.



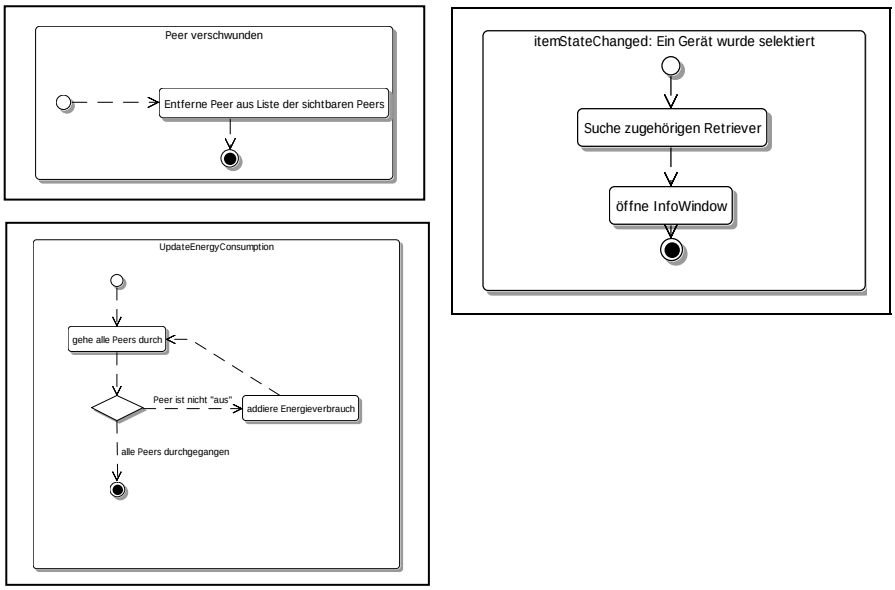
Nachdem die Selbstbeschreibung des Peers durch **MergeUnit** komplett in die Ontologie integriert wurde, wird der Peer abschließend nach seinem aktuellen Status befragt.



Benutzereingaben können über die Buttons, die Eingabezeile oder durch auswählen eines Peers in der Liste erfolgen.



Wird ein Peer ausgewählt, so wird die Methode **ItemStateChanged** aufgerufen. Sie sucht daraufhin den passenden **Retriever** und öffnet das **InfoWindow**. In diesem werden alle **Component**s des Gerätes angezeigt mit ihren jeweiligen **Action**s. Diese können ausgewählt werden, woraufhin der entsprechende Befehl versandt wird.
Abschließend wird ein Peer, wenn er aus dem Netz verschwunden ist, aus der Liste der sichtbaren Peers gelöscht. Sein **Retriever** bleibt jedoch erhalten, da in ihm die Detailparametrierung der Geräte gespeichert wurde. Somit bleiben die zuletzt ausgewählten Aktionen erhalten.



Das **Matching** wurde bereits in Kapitel 6.6 behandelt.

A6. Vollständiges Klassendiagramm der Organic Room Software

Aus Platzgründen findet sich die JavaDoc-Beschreibung des OrganicRoom nur auf der beigelegten CD.