Fachhochschule Aachen
Fachbereich E-Technik und Informationstechnik
Eupenerstr. 70
52066 Aachen

Experimenteller Prototyp zur ontologiebasierten Suche in einem Multi-Agenten-System

Diplomarbeit
von

cand. Diplom Informatiker (FH)
Bernd Müller

Erstprüfer: Prof. Dr. rer. nat. Heinrich Faßbender

Zweitprüfer: Prof. Dr.-Ing. Thomas Ritz

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Die Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, im April 2006

Geheimhaltung

Diese Diplomarbeit darf weder vollständig noch auszugsweise ohne schriftliche Zustimmung des Autors, des betreuenden Referenten vervielfältigt, veröffentlicht oder Dritten zugänglich gemacht werden.

Inhaltsverzeichnis

1 Einleitung

In verteilten Systemen besteht das Problem von Marshalling und Unmarshalling, also das Problem, wie man Daten ohne Verlust über das Netz austauschen kann. Durch die Benutzung von Ontologien kann man dieses Problem umgehen, da das Datenformat vorgegeben ist und somit der Datenaustausch zwischen Knoten eines Netzwerkes geregelt ist. Ebenso beschreiben Ontologien Meta-Informationen über Daten und semantische Zusammenhänge können eingebunden werden.

Bisherige Technologien im Netzbereich von Internet- und Intranet-Architekturen bestehen aus zentral-orientierten Strukturen in Client-Server-Systemen. Ein Service-Provider ist allen Clients im Netzwerk bekannt und wird von ihnen kontaktiert, um einen gewissen Dienst in Anspruch zu nehmen. Die Implementierung solcher Systeme ist von einer zentralen Kontroll-Struktur geprägt. Die Kontroll-Struktur wechselt zwischen Client und Server. Die entsprechende Kontroll-Struktur in der objektorientierten Software-Entwicklung ist ein *Object Request Broker* und es existiert ein solcher für Client und Server, der entsprechend hin- und herwechselt. Also hat ein Programmierer eines solchen zentral-orientierten Systems alle auftretenden Eventualitäten des Wechsels der Kontroll-Struktur zu berücksichtigen.

In einer Peer-to-Peer-Architektur (im Folgenden P2P) repräsentiert jeder Knoten eine gleichwertige Einheit des Systems. Ein Knoten übernimmt nicht die Rolle eines Clients oder Servers. Somit befindet man sich bei der Verwendung einer P2P-Architektur auf einer höheren Abstraktionsebene, weil die Kontroll-Strukturen nicht linear implementiert sind. Jeder Knoten ist eine autonome Einheit und als solche implementiert. Die Kommunikation findet nicht als Frage-Antwort-Szenario statt, sondern jeder Knoten kann gleichzeitig eine Anfrage, eine Antwort oder eine Information verschicken und empfangen.

Kombiniert man diese beiden Technologien P2P-System und Ontologie, ergeben sich daraus ganz neue Möglichkeiten der Entwicklung für Anwendungen im Netzbereich, zum Beispiel zur Implementierung von Suchfunktionen.

Benutzer sind von Suchfunktionalitäten ahängig, um auf Informationen, die sie benötigen, zugreifen zu können. Bisherige Suchmaschinen im Internet sammeln normalerweise Informationen über Dokumente zum Indizieren. Aber Antworten auf Suchanfragen geben keine präzisen Antworten in Bezug zu den Anforderungen des Benutzers, weil es nicht möglich ist, einen Kontext in der Suchanfrage abzubilden. Eine Möglichkeit die Präzision zu erhöhen, wäre den indizierten Informationen noch Meta-Informationen hinzuzufügen. Dies kann mit der Verwendung von Ontologien erzielt werden, welche den Informationen einen semantischen Kontext geben. Ebenso sind die indizierten

Daten solcher herkömmlichen Suchmaschinen statisch auf einem zentralen Server gespeichert. Diese Daten werden periodisch aktualisiert durch Crawlen der indizierten Seiten. Ein verbessertes Szenario wäre es, direkt auf den Daten von verbundenen Peers zu suchen.

Im Rahmen eines wissenschaftlichen Forschungsprojektes der Fachhochschule Aachen wurde ein experimenteller Prototyp entwickelt, um neue Technologien für das Internet-Portal der Stadtverwaltung Aachen zu erproben. Eine Ontologie wurde entworfen, um demonstrativ die Daten des verfügbaren Internet-Portals der Stadtverwaltung zu repräsentieren. Anschließend wurde die Ontologie in verschiedenen Versionen mit unterschiedlichen Instanzen auf Peers eines P2P-Netzwerkes mit Agenten als Knoten verteilt.

Herr Pour-Heidari entwickelte in seiner Diplomarbeit[24] im Rahmen des Forschungsprojekts die Ontologie und ein Client-Server-System zur Suche in dieser Ontologie. Als Fortsetzung dieser Diplomarbeit verteilte ich die Ontologie in unterschiedlichen Versionen mit einer veränderten Anzahl von Instanzen auf Peers in einem Multi-Agenten-System. Der Schwerpunkt dieser Arbeit liegt nicht in der Entwicklung von Ontologien, da diese bereits zur Verfügung standen, sondern darin Ontologien in einem Multi-Agenten-System einsetzen zu können.

Die Entwicklung dieses **experimentellen Prototyps zur ontologiebasierten Suche in einem Multi-Agenten-System** ist Gegenstand meiner Diplomarbeit.

Zunächst wird auf bisherige Formen des Suchens im Internet eingegangen, um daraufhin die neuen Technologien Agenten und Ontologien zu beschreiben. Anschließend wird das verwendete Framework erläutert und veranschaulicht an einer beispielhaften Implementierung eines Agentensystems. Im Anschluß daran wird die verwendete Bibliothek beschrieben mit der man Ontologien und Agenten zusammen implementieren kann. Nachdem die verwendeten Technologien, die Frameworks und die Bibliothek beschrieben wurden, wird die Entwicklung des Prototyps vorgestellt. Abschließend werden weitere Ausblicke gegeben und Einsatzmöglichkeiten des Prototypen aufgezeigt.

Inhaltlich ist die Diplomarbeit in folgende Kapitel gegliedert:

- **Kapitel 2**: In diesem Kapitel werden die bisherigen Suchmaschinen im Internet vorgestellt, wie diese grob umschrieben funktionieren, und welche Architekturmodelle ihnen zu Grunde liegen.

- **Kapitel 3**: In diesem Kapitel werden die Begriffe der Technologien, die dieser Diplomarbeit zu Grunde lagen, definiert.

- **Kapitel 4**: In diesem Kapitel wird das verwendete JADE-Framework vorgestellt und anhand eines Beispiels verdeutlicht.

- **Kapitel 5**: In diesem Kapitel wird die Bibliothek AgentOWL mit ihrer Demo erläutert.
- **Kapitel 6**: In diesem Kapitel wird die Entwicklung des experimentellen Prototyps beschrieben.
- **Kapitel 7**: In diesem Kapitel wird ein Fazit zur Entwicklung des Prototyps gezogen.
- **Kapitel 8**: In diesem Kapitel werden Ausblicke und Einsatzgebiete des Prototypen aufgezeigt.

In Abbildung 1 sind die verwendeten Tools und Frameworks dargestellt. Mit Jade werden die Agenten entwickelt, die auf die Bibliothek von AgentOWL zugreifen. AgentOWL verwendet widerum Jena, um auf RDF/OWL-Dateien zuzugreifen, die mit Protégé erstellt werden.

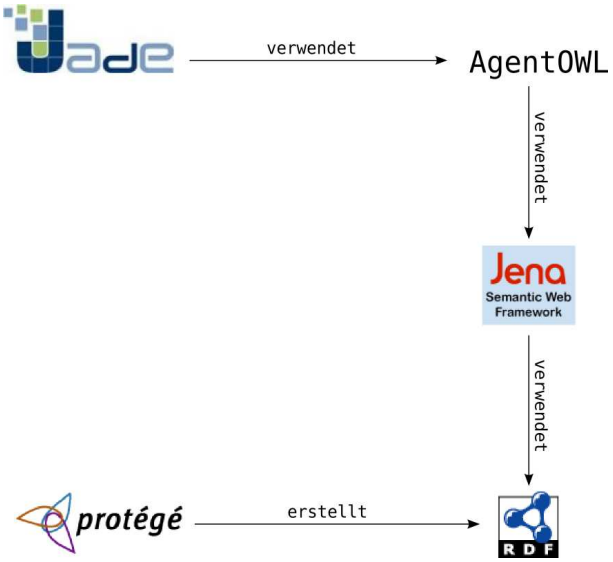

Abbildung 1: Die verwendeten Tools

2 Das Suchen im Internet

Bei der Betrachtung von Suchoptionen im Internet gehe ich nur auf die etablierten Technologien seit der Freigabe des **WWW** 1991 durch Tim Berners-Lee ein. Immer mehr in HTML geschriebene Dokumente finden sich im Internet. Musik und Videos stehen in digitaler Form zur Verfügung. Es verbreiten sich immer mehr Dienstanbieter, ob dies Mailservice-Anbieter, Internet Service Provider, Webshops oder Webspace-Anbieter sind. Um sich in all diesen Informationen zurecht zu finden, wurden Suchmaschinen entwickelt, die sich in dem großen Datenbestand zurecht finden, und dem Benutzer aufbereitete Informationen zurückgeben. Im diesem Kapitel gebe ich eine grobe Übersicht über die historische Entwicklung von Technologien des Suchens im Internet. Die Liste erhebt keinen Anspruch auf Vollständigkeit.

2.1 Geschichte des Suchens im Internet

Als erste richtungsweisende Suchmaschine galt **Gopher**, welche 1991 an der University of Minnesota von Paul Lidner und Mark P. McCahill entwickelt wurde. **Gopher** baut auf das in RFC 1436 beschriebene **Gopher**-Protokoll auf. Es bietet im Gegensatz zu in HTML dargestellten Seiten ein automatisch generiertes Menü an, das die im aktuellen Verzeichnis dargestellten Dateien visuell anders darstellt. Des weiteren kann eine Webseite einen speziell für **Gopher** entwickelten Link enthalten, der auf eine andere Webseite verweist.

1994 wurde **Lycos** von Michael Mauldins entwickelt, das erstmals die Häufigkeit von Suchbegriffen, sowie auch ihre Nähe zueinander, betrachtete. 1995 erschienen die ersten kommerziellen Suchmaschinen, wie zum Beispiel **Infoseek**, **Architext** und **Altavista**.

Im folgenden Jahr entstanden dann auch die ersten Metasuchmaschinen, welche mehrere andere Suchmaschinen nutzen, Suchanfragen an sie delegieren und aufbereitet an den Benutzer zurück geben. Metasuchmaschinen sind zum Beispiel **Metager** und **Metacrawler**.

1995 entwickeln Larry Page und Sergey Brin die Suchmaschine **BackRub**, dem Vorläufer von **Google**. **Google** geht 1998 im Betastadium ans Netz und etabliert sich in den folgenden Jahren als erfolgreichste Suchmaschine des Internets.

1998 wurde die Musiktauschbörse **Napster** von Shawn Fanning programmiert, um leichter über das Internet MP3-Musikdateien austauschen zu können. Mit ihm konnte (wurde abgeschaltet aufgrund zahlreicher Copyright-Verletzungen) der Benutzer Musiktitel suchen und herunterladen.

2000 wurde das **Gnutella**-Protokoll von Justin Frankel definiert und in der Betaversion ins Netz gestellt. Frankels Arbeitgeber AOL zwang ihn jedoch

nach kurzer Zeit das Projekt aufzugeben und es nicht weiter zu veröffentlichen. Das Programm war zu diesem Zeitpunkt jedoch schon weit verbreitet und verfügbar über fremde Webseiten und Tauschbörsen. Einige Zeit später gelang es einer Gruppe unabhängiger Entwickler das **Gnutella**-Protokoll zu entschlüsseln und ihre Ergebnisse zu veröffentlichen. Daraufhin wurden zahlreiche weitere Programme für das **Gnutella**-Netzwerk entwickelt, die auch den Leistungsumfang des Netzwerkes deutlich erweiterten. Derzeit zählt **Gnutella** ca. 2,2 Millionen Nutzer.

2001 wurde die Software **KaZaA** veröffentlicht, die von Niklas Zennström und Janus Friis entwickelt wurde. **KaZaA** ist eine Internet-Tauschbörse mit dem Nutzer aus aller Welt Dateien aller Art tauschen können, wie zum Beispiel MP3s, Videos, Textdokumente und Bilder. **KaZaA** hatte im April 2003 4,4 Millionen Benutzer dessen Zahl aber nach einem Verfahren vor dem Europäischen Markenamt stark sank.

2001 wurde **BitTorrent**, eine weitere Tauschbörse, entwickelt. Nach einer Studie von Cachelogic.com im Zeitraum von Januar 2004 bis Juli 2004 soll der Netzwerkverkehr, der mit **BitTorrent** verursacht wird, bereits 1/3 des gesamten Netzwerkverkehrs des Internets ausmachen. Insgesamt macht der Netzwerkverkehr von Tauschbörsen die Hälfte des Datenaufkommens im Internet aus.

2002 entstand das **eMule**-Projekt, was **KaZaA** sehr ähnelt. Im Dezember 2005 zählte das **eMule**-Netzwerk 3,3 Millionen Nutzer[1].

2.2 Technologien des Suchens

Die verschiedenen Suchmaschinen und Tauschbörsen verwenden unterschiedliche Ansätze der Implementierung eines Architekturmodells. Sie folgen einem **Client-Server**-Modell, einem **P2P**-Modell oder einem **hybriden** Modell, was eine Kombination aus den beiden anderen darstellt. Je tiefer man ins Detail der Implementierung einer Architektur geht, desto weniger lässt sich differenzieren, welches allgemeine Modell tatsächlich verfolgt wird. Die Abstraktionsebene, auf der man solche Modelle betrachtet, spielt also eine entscheidende Rolle.

2.3 Client-Server-Modell

Das Client-Server-Modell findet sich in klassischen Suchmaschinen, wie **Gopher** und **Google** wieder. Eine Suchmaschine des Client-Server-Modells liefert Informationen über Dokumente, die im WWW zur Verfügung gestellt

[1] http://slyck.com

sind. Um Informationen bereit zu stellen, müssen die Dokumente jedoch erfasst und die Informationen aufbereitet werden. Dies erfolgt in der Regel in den folgenden Schritten:

- Sammeln und Erfassen von Informationen und Dokumenten aus dem WWW,
- Analysieren und Indexieren der gesammelten Daten,
- Ablegen der Daten gemäß dem erstellten Index,
- Überprüfen von bereits erfassten Dokumenten auf Änderungen,
- Ausgabe von Suchergebnissen sortiert nach ihrer Relevanz bezüglich der Suchanfrage (Ranking).

Die indizierten Informationen, die nach einem Ranking abgespeichert sind, werden nach dem Client-Server-Prinzip dem Benutzer zur Verfügung gestellt. Der Benutzer setzt über einen Browser, dem Client, eine Suchanfrage ab und erhält als Ergebnis vom Server eine Liste mit nach Treffergenauigkeit sortierten URLs.

2.4 P2P-Modell

Bei einer P2P-Architektur stellt jeder Knoten eine gleichberechtigte Einheit im System dar. Ein Knoten übernimmt nicht eindeutig die Rolle eines Dienstnehmers oder Dienstanbieters. Das stellt ein höheres Abstraktionsniveau dar, in der Hinsicht, dass man die Kontroll-Struktur des Gesamtsystems nicht linear implementiert, sondern jeder Knoten eine in sich geschlossene Einheit darstellt und auch so implementiert wird. Die Kommunikation zwischen Knoten findet nicht als Anfrage- und Antwort-Szenario statt, sondern jeder Knoten kann gleichzeitig

- eine Anfrage stellen (Client),
- eine Antwort geben (Server) oder
- eine Information versenden.

In einer reinen Implementierung des P2P-Modelles besteht die Architektur aus gleichwertigen Knoten, die miteinander kommunizieren und keine spezielle Rolle, ob als Client oder als Server, einnehmen. Ein P2P-System lässt sich wie folgt charakterisieren:

- Es gibt keine zentrale Datenbank, jeder Peer stellt einen Teil der vorhandenen Informationen zur Verfügung. Kein Peer verwaltet (oder kennt) den Gesamtbestand.
- Es gibt keine zentrale Instanz, die Interaktionen steuert oder koordiniert,
- Peers sind autonom,
- Kein Peer hat (notwendigerweise) einen Überblick über das Gesamtsystem. Jeder Peer kennt nur die Peers, mit denen er interagiert,
- Das Verhalten des Systems ergibt sich dynamisch aus der Kombination der Interaktionen zwischen den Peers,
- Peers, Verbindungen und Informationen sind nicht verlässlich.

Ein reines P2P-Modell zum Suchen existiert in der Regel nicht und es herrscht meistens eine Mischform, also eine Kombination aus Client-Server- und P2P-Modell.

2.5 Hybrides Modell

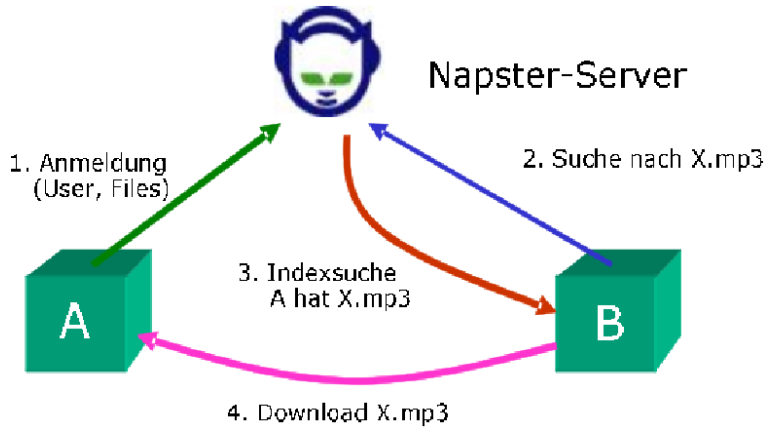

Abbildung 2: Architektur und Kommunikationsmodell von Napster[1]

Beispielhaft möchte ich hier das Architektur-Modell von Napster beschreiben, was so oder in anderer Form von anderen Tauschbörsen ebenso implementiert ist und einem hybriden Modell unterliegt. Technisch gesehen handelt es sich bei Napster um ein relativ simples P2P-System: Der Napster-Server

hält eine zentrale Datenbank der angebotenen MP3-/ WMA-Musikdateien, die Dateinamen mit IP-Adressen von Peers verknüpft. Clients (man beachte die Terminologie) melden sich bei diesem Server an und schicken eine Liste von Dateien, die sie zur Verfügung stellen. Neubenutzer müssen dazu zuerst einen Account beim Napster-Server einrichten. Der Vorgang ist in Abbildung 2 dargestellt.

Jeder Client kann Suchanfragen an den Napster-Server stellen und erhält als Suchergebnis eine Liste an Clients, die der Suchanfrage entsprechende Dateien anbieten. Aus dieser Liste kann ein Client ausgewählt werden, um von diesem die gewünschte(n) Datei(en) herunterzuladen.

Bei Napster handelt es sich also um kein reines P2P-System, sondern vielmehr um eine Kombination des Client-Server- und des P2P-Ansatzes. Das Kommunikationsprotokoll von Napster wurde nie publiziert. Es existieren lediglich Protokolldefinitionen, die „re-engineered“ wurden. Aufgrund der zentralen Datenbank von Napster (auch wenn sie in Wirklichkeit auf vielen Servern repliziert war), stellt dies einen Flaschenhals dar, der die Skalierbarkeit von Napster erheblich beschränkt. Der Vorteil dieser Architektur ist allerdings, dass dies eine einfache Kontroll- und Steuer-Möglichkeit bietet, was in vielen verteilten Systemen notwendig ist bzw. gewünscht wird. Außerdem gibt es mit einer solchen Architektur geringere Probleme bezüglich der Konsistenz des (zentralen) Index als bei verteilten Indexansätzen. [1]

3 Technologien

In Kapitel 2 wurde die Ausgangssituation beschrieben. In diesem Kapitel werden die Technologien aufgezeigt mit denen eine Verbesserung erreicht werden kann. Bei der Erstellung meiner Diplomarbeit stellte sich mir die Frage, was eigentlich ein Agent ist und was ihn von anderen Paradigmen in der Softwareentwicklung unterscheidet. Bei meinen Recherchen stellte sich heraus, dass es nicht wirklich eine einheitliche Definition gibt, man aber in etwa umreißen kann, was an einem Agenten prägnant ist und wie es sich von anderen Paradigmen, wie zum Beispiel der Objektorientierten Softwareentwicklung unterscheidet. Agenten sind kleine, eigenständige Programme, die jeweils eine Funktion selbständig erfüllen und mit anderen Agenten oder Benutzern kommunizieren und interagieren. Im Gegensatz zur objektorientierten Softwareentwicklung in der ein gesamtes System entwickelt wird, um eine gewisse Problemstellung zu lösen, stellt der Ansatz in der Entwicklung von Agentensystemen den Entwurf von autonomen Softwareeinheiten dar, die jedes für sich eine Problemstellung lösen und ein bestimmtes Ziel verfolgen. Bei der objektorientierten Softwareentwicklung spiegelt sich zum Beispiel das Prinzip der Kapselung darin wieder, eine Methode als privat zu deklarieren. Somit kann ein Objekt anderen Objekten den Zugriff auf gewisse Methoden verbieten und hat somit die Kontrolle über seinen internen Zustand. Es hat aber nicht die Kontrolle über sein Verhalten. Man kann zwar eine Steuerung programmieren, wann und wie ein anderes Objekt auf eine Methode zugreifen darf, was aber nicht als Prinzip im Entwurf von objektorientierten Programmiersprachen zu Grunde lag.

Bei Agenten jedoch ist ein **Entwurfsmuster**, das **Verhalten**, zu implementieren. Ein Agent, der auf die Ressource eines anderen Agenten zugreifen will, stellt eine Anfrage und es obliegt dem angefragten Agenten, ob er den Zugriff auf diese Ressource erlaubt oder nicht. Im Folgenden versuche ich genauer zu definieren, was ein Agent ist, wie er sich von anderen Technologien unterscheidet und was sein Zweck ist.

3.1 Definition eines Agenten

Der Begriff Agent kommt ursprünglich aus dem Forschungsbereich der Künstlichen Intelligenz. Jedoch ist es selbst dort nicht einheitlich definiert, was unter diesem Begriff zu verstehen ist. Eine einheitliche Definition im Forschungsbereich der Künstlichen Intelligenz und in der Entwicklergemeinde von Agentensystemen gibt es nicht. [5] Teilweise entwarf jede Arbeitsgemeinschaft ihre eigene Definition darüber, was unter einem Agenten zu verstehen ist. Um einen Umriss über die Kernaspekte eines Agenten zu geben, führe ich

einige Definitionen auf, die später in diesem Kapitel anhand eines Beispiels, dem Hotplug-Agentensystem unter Linux, veranschaulicht werden:

- **MuBot Agent:** *Der Term Agent wird verwendet, um zwei orthogonale Konzepte zu repräsentieren. Das erste ist die Fähigkeit des Agenten autonom zu handeln. Das zweite ist die Fähigkeit in bestimmten Bereichen logisch zu denken. [6]*

- **AIMA Agent:** *Als Agent kann alles angesehen werden, was seine Umgebung durch Sensoren wahrnimmt und durch Effektoren diese Umgebung beeinflusst. [7]*

- **Maes Agent:** *Autonome Agenten sind Computersysteme, die in einer dynamischen und komplexen Umgebung vorkommen, diese Umgebung wahrnehmen und darin autonom agieren, sowie dabei eine Gruppe von Zielen oder Aufgaben erfüllen für die sie entwickelt wurden. [8]*

- **KidSim Agent:** *Man kann einen Agenten definieren als eine persistente Softwareeinheit, die dem Erreichen einer bestimmten Aufgabe dient. 'Persistent' unterscheidet Agenten von Unterroutinen; Agenten haben ihre eigenen Ideen, wie sie ihre Aufgaben erfüllen, sie haben ihre eigene Planung. 'Das Erreichen einer bestimmten Aufgabe' unterscheidet die Agenten von allen multifunktionalen Applikationen; Agenten sind üblicherweise wesentlich kleiner. [9]*

- **Hayes-Roth Agent:** *Intelligente Agenten erfüllen fortschreitend drei Funktionen:*

  1. *Wahrnehmen von dynamischen Zuständen in der Umgebung,*
  2. *Agieren, um Zustände in der Umgebung zu beeinflussen,*
  3. *Schlussfolgern, um Wahrnehmungen zu interpretieren, Probleme zu lösen, Rückschlüsse zu machen und Handlungen zu entwickeln.*

  [10]

- **Jennings Agent:** *... eine Hardware oder (üblicher) ein softwarebasiertes Computersystem, dass die folgenden Eigenschaften hat:*

  1. *Autonomie: Agenten handeln ohne direkte Eingriffe durch Menschen oder anderem und haben eine Art Steuerung ihrer Aktionen und internen Zustände.*

2. *Soziale Fähigkeiten: Agenten interagieren mit anderen Agenten (und vielleicht Menschen) über eine Art Agentenkommunikationssprache.*
3. *Reaktivität: Agenten nehmen ihre Umgebung wahr (was die reale Welt, ein Benutzer über ein graphisches Interface, eine Zusammenstellung anderer Agenten, das Internet oder vielleicht eine Kombination aus allem sein kann) und antworten in zeitlicher Abhängigkeit auf auftretende Änderungen.*
4. *Proaktivität: Agenten agieren nicht einfach als Antwort auf ihre Umgebung, sondern sind fähig zielgerichtetes Verhalten zu entwickeln, indem sie die Initiative ergreifen.*

[11]

- **Brustolini Agent:** *Autonome Agenten sind Systeme, die fähig sind autonome und gezielte Aktivitäten in der realen Welt durchzuführen. [12]*

Im weiteren Verlauf meiner Diplomarbeit benutzte ich die in [13] beschriebene Definition eines Agenten, die an die Definition des Jenning Agenten angelehnt ist:

> *Ein Agent ist ein Computersystem **eingebettet** in eine gewisse Umgebung, das fähig ist **flexible** und **autonome** Aktionen durchzuführen, um seine Ziele zu erreichen, für die es entworfen wurde.*

Die drei Kernkonzepte in dieser Definition sind:

- **Einbettung:** Dies bedeutet in diesem Kontext, dass der Agent Sinneswahrnehmungen aus seiner Umgebung erhält und dass er Aktionen durchführen kann, die seine Umgebung in einer gewissen Weise ändert.
- **Flexibilität:** Unter Flexibilität ist zu verstehen, dass das Computersystem:
  - **reagierend** ist: Agenten sollten ihre Umgebung wahrnehmen und auf Änderungen der Umgebung in zeitlichem Kontext antworten;
  - **proaktiv** ist: Agenten sollten nicht nur agieren als Antwort zu ihrer Umgebung, sie sollten opportunistisches, zielgerichtetes Verhalten zeigen und die Initiative ergreifen, wenn es angebracht ist;

  - **sozial** ist: Agenten sollten in der Lage sein, mit anderen Agenten oder Benutzern zu interagieren, wenn es angebracht ist, damit sie ihre Probleme lösen und anderen bei ihren Aktivitäten helfen.

- **Autonomie:** Darunter ist zu verstehen, dass das Computersystem in der Lage ist, ohne direkten Eingriff des Benutzers (oder anderer Agenten) zu agieren und dass es die Steuerung über seine eigenen Aktionen und seinen internen Zustand hat.

3.2 Definition eines agentenbasierten Systems

Unter einem **agentenbasierten System** versteht man schlicht, dass die Schlüsselabstraktion Agent bei der Entwicklung verwendet wurde. Prinzipiell kann ein **agentenbasiertes System** von der Konzeptionierung her unter dem Term Agent entwickelt worden sein, jedoch auch ohne die Verwendung von Software-Strukturen eines Agenten entwickelt worden sein. So ist es zum Beispiel möglich, in einer objektorientierten Umgebung alles ohne die Verwendung von Objekten rein prozedural zu programmieren. Dies wäre jedoch ungewöhnlich und im schlimmsten Falle sogar kontraproduktiv. Eine ähnliche Situation gibt es bei der Agententechnologie. Man kann das System unter der Abstraktion Agent designen und bei der Implementierung den objektorientierten Ansatz verfolgen. [13] Ein **agentenbasiertes System**, dass von der Konzeptionierung her unter dem Term Agent entwickelt wurde, aber nicht die Software-Strukturen eines Agenten bei der Entwicklung nutzte, steht jedoch weiterhin unter der Pragmatik Agenten mit den drei Kernkonzepten *Einbettung*, *Autonomie* und *Flexibilität* als Komponenten zu haben. Unter einem **agentenbasierten System** versteht man, grob beschrieben, ein System, dessen Komponenten aus Agenten bestehen.

3.3 Definition eines Multi-Agenten-Systems (MAS)

Bei einem **Multi-Agenten-System** oder kurz **MAS** handelt es sich um ein System aus mehreren gleichartigen oder unterschiedlich spezialisierten handelnden Einheiten, die kollektiv ein Problem lösen. Die Einheiten sind in der Regel Softwareprogramme, also Software-Agenten. [14] Bei der objektorientierten Softwareentwicklung versteht man den Entwicklungsprozess als ein Vorgehen das gesamte zu programmierende System in chronologischer Reihenfolge zu erfassen und zu implementieren. Ein **MAS** jedoch besteht aus einzelnen agentenbasierten Systemen.

3.4 Hotplug-Agentensystem unter Linux

Um die Definitionen von Agenten und Multi-Agenten-Systemen darzustellen, beschreibe ich hier exemplarisch das unter Linux als MAS implementierte Hotplug-System. Das Hotplug-System besteht als Open-Source-Projekt innerhalb der Linux-Community und ist in fast jeder aktuellen Linux-Distribution vorzufinden. Eine komplette Dokumentation über das Projekt gibt es nicht, sondern nur die üblichen Man- und Info-Pages, wie bei allen Softwarepaketen unter Linux. Das Projekt läuft über Sourceforge und ist unter http://linux-hotplug.sourceforge.net erreichbar.

Beim Anschließen von neuer Hardware oder Änderung bestehender Hardware muss vom Betriebssystem der entsprechende Treiber geladen oder geändert werden. Da jedoch die Auswahl, das Laden und auch das Konfigurieren des Treibers dem normalen Nutzer nicht zugemutet werden können, wurden diese Prozesse automatisiert. Zwei Fälle beim Laden von Treibern, die vom Betriebssystem gehandhabt werden, sind zu unterscheiden:

- einen Treiber laden, wenn eine Applikation auf eine Hardware zugreifen möchte, deren Treiber noch nicht geladen ist: **modprobe**
- einen Treiber laden, wenn eine neue Hardware erkannt wird: **hotplug**

Das sogenannte **hotplug** wird unterstützt beim Suchen eines Treibers für ein neues Gerät vom Kernel. Identifiziert der Kernel oder eines seiner Subsysteme ein neues Gerät, wird das **hotplug**-Programm aufgerufen. Als Parameter erhält es die Hardware-respektive Geräte-Identifikation übergeben. Aufgrund dieser Geräte-Identifikation kann **hotplug** zunächst das Subsystem (z.B. PCI oder USB) ermitteln, über welches das Gerät angekoppelt ist. Für jedes Subsystem existiert ein Shell-Skript, das aufgerufen wird und den zu ladenden Gerätetreiber bestimmt. Der gesamte Vorgang ist in Abbildung 3 dargestellt. Die Skripte selbst befinden sich unterhalb des Verzeichnisses

```
/etc/hotplug/
```

und werden nach dem Subsystem und dem Zusatz

```
.agent
```

benannt:

```
pci.agent
```

oder

```
usb.agent.
```

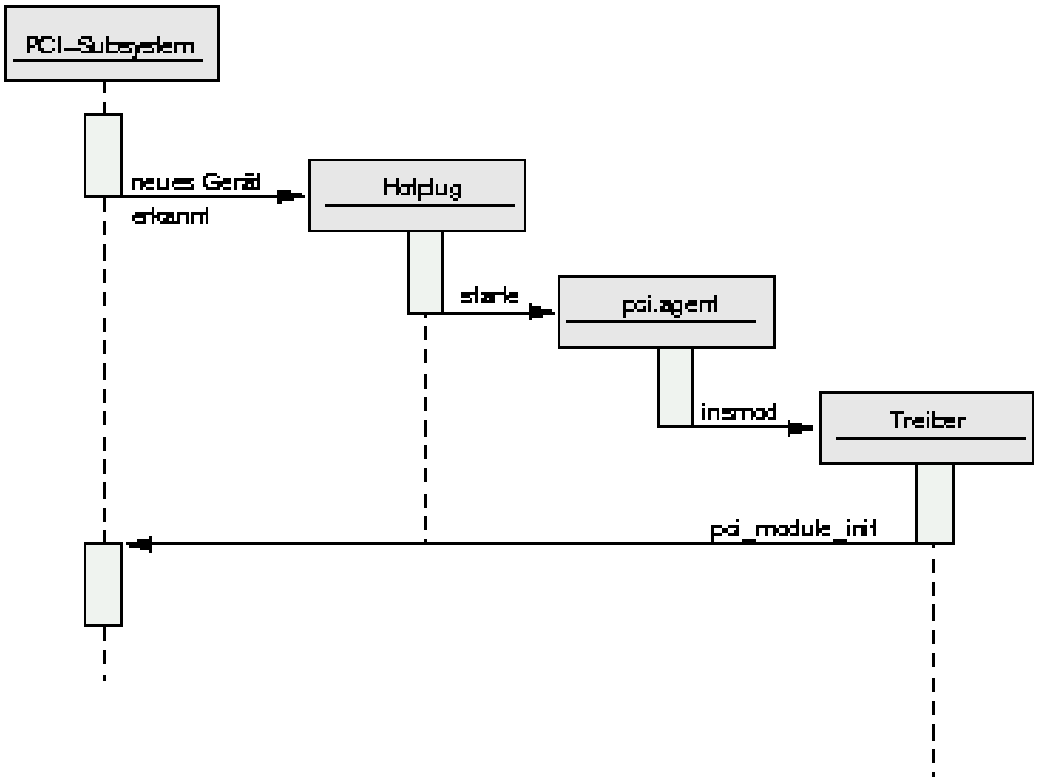

Abbildung 3: Hotplug-Mechanismus

Die Bestimmung des Gerätetreibers erfolgt ebenfalls aufgrund der Geräte-Identifikation. Sämtliche im System bekannten Geräte sind dazu in einer Tabelle zusammen mit dem Namen des zugehörigen Treibers aufgelistet. Diese Tabelle wird bei der Generierung des Kernels automatisch erstellt. Jeder Treiber muss seinen Beitrag zur Erstellung der Tabelle dergestalt leisten, dass er die Geräte, für die er zuständig ist, bekannt gibt. Hierzu dient das Makro

```
MODULE_DEVICE_TABLE.
```

Diesem werden zwei Parameter übergeben. Der erste Parameter gibt das Subsystem an. Der zweite Parameter ist die Liste mit den unterstützten Geräte-Identifikationen. Das Makro sorgt dafür, dass die Geräte-Identifikation vom Linker so zum Treiber gebunden wird, dass sie vom Programm **depmod** ausgewertet werden können. **depmod** erstellt aus dieser Information unterhalb des Verzeichnisses

```
/lib/modules/kernelversion/
```

für jedes Subsystem eine Datei, die mit dem String

```
modules
```

beginnt und eine Erweiterung hat, die mit dem Namen des Subsystems und dem String

```
map
```

endet:

```
modules.pcimap
```

oder

```
modules.usbmap.
```

Exportiert ein Treiber die von ihm bedienten Geräte nicht auf diese Weise, kann eine Tabelle im Hotplug-Verzeichnis

```
/etc/hotplug/
```

immer noch von Hand erstellt werden. Solche Listen heißen dann beispielsweise

```
usb.handmap
```

oder

```
pci.handmap.
```

Darüber hinaus gibt es aber auch die entgegengesetzte Situation, dass ein Modul nicht geladen werden soll, obwohl ein entsprechender Eintrag in den

```
Maps
```

vorhanden ist. In diesem Fall wird das Modul in die Datei

```
blacklist
```

(ebenfalls im Hotplug-Verzeichnis zu finden) eingetragen. [15] Alle Agenten, die Geräte initialisieren müssen, versuchen ein Treibermodul für das Gerät zu finden. Hierzu werden einige Modulmaps ausgewertet. Alle Agenten schauen zuerst in eine Datei

```
/etc/hotplug/~.handmap
```

und nur wenn dort nichts gefunden wurde, durchsuchen sie

```
/lib/modules/kernelversion/modules.~map
```

Nach der Erkennung eines neuen Gerätes wird im Kernel ein Ereignis ausgelöst. Der Kernel startet anschließend den eigentlichen Agenten. Der Agent ist als Shell-Skript implementiert und sucht anhand der Geräte-Identifikation nach dem entsprechenden Treiber in der Tabelle. Nachdem der passende Treiber gefunden wurde, wird er geladen und das Gerät ist für das System verfügbar. Ebenso ist der Agent dafür zuständig nach Entfernen des Gerätes den Treiber auch wieder zu entladen.

Zusammengefasst lassen sich bei diesem Agentensystem die drei Kernkonzepte herausstellen:

- **Einbettung:** Der entsprechende Hotplug-Agent ist an den Kernel gekoppelt, der im Falle eines entsprechenden Ereignisses den Agenten startet.
- **Autonomie:** Nach dem Start handelt der Hotplug-Agent eigenständig und kümmert sich unabhängig von anderen Programmen, wie zum Beispiel dem Kernel, um das Laden des entsprechenden Treibers anhand der Gerätetabelle.
- **Flexibilität:**
  - **reagierend**: Wird ein Gerät, dessen Treiber geladen ist, wieder entfernt, kümmert sich der Agent darum, dass der entsprechende Treiber auch wieder entladen wird.
  - **proaktiv**: Nach dem Erkennen eines neuen Gerätes kümmert sich der Hotplug-Agent um das Starten des passenden Treibers.
  - **sozial**: Der Hotplug-Agent interagiert mit dem Kernel, welcher den Agenten nach entsprechenden Ereignissen startet oder beendet.

3.5 Vergleich von Agenten mit Webservices

Nachdem mögliche Definitionen beschrieben wurden, könnte man meinen, ein Agent sei ein Webservice oder ein Webservice sei ein Agent. Den Unterschied zwischen diesen beiden Technologien beschreibe ich im folgenden. Ein Webservice beschreibt auf einem höheren Abstraktionslevel als Agenten den Datenaustausch zwischen verschiedenen Systemen über das Netz. Literatur findet man hierzu unter [3]. Agenten sind die konkrete Implementierung eines modellierten Webservices. Ein Webservice stellt einen Sonderfall eines Agentensystems dar. Er beschreibt, dass eine Ressource einen Dienst über ein Verzeichnis im Netz propagiert, damit ein anderer auf diese Ressource

über das Verzeichnis Zugang findet und anschließend mit der Ressource über ein vordefiniertes Schema, einer Schnittstelle, kommunizieren kann, um die gewünschten Resultate zu erhalten.

3.5.1 Definition eines Webservices

Ein Webservice ist ein Softwaresystem, was entworfen wurde, um vollständig kompatible Maschineninterakionen über ein Netzwerk zu unterstützen. Es hat ein Interface beschrieben in einem maschinenlesbaren Format (zum Beispiel Webservice Description Language). Andere Systeme interagieren mit einem Webservice in einer vorgeschriebenen Art beschrieben durch die Verwendungsweise von SOAP (Simple Object Access Protocol). Normalerweise werden diese Nachrichten über HTTP mit einer XML-Serialisierung in Verbindung mit anderen webverwandten Standards versendet. [4]

3.5.2 Webservices und Agenten

Ein Webservice ist eine abstrakte Notation, die von einem konkreten Agenten implementiert werden muss. Der Agent ist das konkrete Stück Software oder Hardware, welches Nachrichten versendet und empfängt, während der Webservice die Ressource ist, die durch eine abstrakte Menge von Funktionalitäten charakterisiert ist. Um diesen Unterschied zu veranschaulichen, kann man einen bestimmten Webservice an einem Tag mit einem Agenten implementieren (geschrieben in einer bestimmten Programmiersprache) und an einem anderen Tag den gleichen Webservice mit einem anderen Agenten implementieren (geschrieben in einer anderen Programmiersprache) mit der gleichen Funktionalität. Auch wenn sich die Agenten geändert haben, bleibt der Webservice, der von ihnen implementiert wird, der gleiche. [4] Jede Implementierung eines Webservices ist ein Agent, aber jeder Agent ist nicht immer die Implementierung eines Webservices.

3.6 Vergleich von Agenten mit der objektorientierten Softwareentwicklung

Die Datenrepräsentation in der Softwareentwicklung beim Übergang von der prozeduralen Programmierung zur objektorientierten Programmierung, stellte einen Paradigmenwechsel von einer technozentrischen Perspektive zu einer anthropozentrischen Perspektive dar, also den Wechsel von der technisch-orientierten Sichtweise zur menschlich-orientierten Sichtweise [19]. Die Funktionalität stand nicht mehr im Vordergrund, sondern der Anwendungsfall.

Daten wurden als Objekte modelliert und nicht mehr an die optimale Datenspeicherung orientierte Datenstrukturen.

Die Verwendung von Agenten als Paradigma in der Softwareentwicklung stellt einen Wechsel von der objektorierten Programmierung zur zielorientierten Programmierung dar. Es steht nicht mehr das statisch existente Objekt aus der Realität im Vordergrund, sondern die Rolle eines Akteurs, der modelliert wird.

Der Programmablauf innerhalb eines objektorientierten Systems ist durch eine stetige Kontroll-Struktur gekennzeichnet. Ein Objekt gibt innerhalb des Programmablaufs die Kontrolle an ein anderes Objekt weiter. So wechselt zum Beispiel bei einer Client-Server-Architektur die Kontrollstruktur zwischen Client und Server hin und her. Die jeweilige Kontroll-Struktur ist ein ORB[2] und es existiert jeweils ein Client- und Server-ORB. Der Programmierer hat also bei der Programmierung eines solchen objektorientierten Systems alle Eventualitäten des Wechsels der Kontroll-Struktur zu berücksichtigen.

Bei einem Agentensystem bildet jeder einzelne Agent im System eine autonome Einheit, das heißt, dass er jederzeit die volle Kontrolle über seinen internen Zustand hat und kein anderer Agent. Möchte ein Agent eine Dienstleistung von einem anderen Agenten in Anspruch nehmen, kann er nur eine Anfrage an den anderen Agenten stellen, damit dieser ihm die Dienstleistung zur Verfügung stellt. Ob der andere Agent die Dienstleistung nun zur Verfügung stellt oder nicht, wird intern von diesem verarbeitet.

Für den Entwickler bedeutet dies, jeweils einen einzelnen Agenten als in sich geschlossenes System anzusehen. Die Kommunikation mit anderen Agenten findet über Nachrichtenaustausch statt.

3.7 Agenten in Netz-Architekturen

Bestanden bislang die Architekturen in Netzwerken größtenteils aus Client-Server-Architekturen, bedeutet die Nutzung eines Agentensystems den Wechsel zu einer **P2P-Architektur**. Dienste werden nicht mehr zentral von einem Server im Netzwerk den Clients zur Verfügung gestellt, sondern von Agenten dezentral angeboten. Sie sind untereinander verbunden. Bei einem Sonderagenten, der einen Verzeichnisdienst verwaltet, können Agenten ihre Dienste propagieren und nach bestimmten Diensten suchen, die von Agenten angeboten werden.

Bei einer Client-Server-Architektur existieren die Rollen des Dienstanbieters und eines Dienstnehmers. Der Dienstnehmer stellt eine Anfrage an den Dienstanbieter, eine gewisse Information zu bearbeiten oder zur Verfügung

[2] Object Request Broker

zu stellen. Der Dienstanbieter verarbeitet die Information oder stellt eine von ihm verwaltete Information bereit. Um die neue Information dem ursprünglichen Dienstnehmer zu übergeben, wechselt der Dienstanbieter in die Rolle des Dienstnehmers und der Dienstnehmer in die Rolle des Dienstanbieters. Zur Abgabe der ursprünglichen Anfrage werden die Rollen also vertauscht und es existieren jeweils die Positionen des Fragenden und Antwortenden. Die Kontroll-Struktur ist hier, wie in der objektorientierten Softwareentwicklung, linear.

Der Übergang von einer solchen zentralen Client-Server-Architektur zu einer dezentralen P2P-Architektur, ermöglicht das gesamte System flexibel und dynamisch zu halten. Agenten können dem System beitreten und das System wieder verlassen, ohne dass die Funktionalität des gesamten Systems gefährdet ist. Jeder Agent bildet bei dieser Architektur eine gleichberechtigte Komponente.

3.8 Definition von Ontologien

Um das Kapitel über die verwendeten Technologien abzuschließen, gehe ich noch kurz auf Ontologien ein, die sehr ausführlich bereits in der Arbeit von Herrn Pour-Heidari[24] dargestellt wurden.

Der Begriff Ontologie kommt ursprünglich aus dem Bereich der Philosophie und ist die Leere des Seienden. Dabei geht es darum, die grundlegenden Eigenschaften von Dingen zu beschreiben. „Was ist ein Stuhl?“ oder anders gesagt, „Was ist die Idee eines Stuhles?“. Was stellt sich der Mensch darunter vor, wenn er das Wort Stuhl gebraucht? Es ist nicht die eigentliche Instanz der Klasse Stuhl, sondern es sind die Eigenschaften, die einen Stuhl ausmachen und zu dem machen, was er ist.

In der Informatik versteht man unter einer Ontologie die Konzeptionierung einer hierarchischen Struktur von Begriffen (in der Anwendung meistens nicht Begriffe, sondern Klassen). Die bekannteste Definition lautet **Spezifikation einer Konzeptionierung**. Zusätzlich enthalten Ontologien Inferenz- und Integritäts-Regeln. Es ist also eine Art Modellierung einer Klassenhierarchie. Bei einer Ontologie kommt es konkret nicht darauf an, wie sie implementiert ist, wie die Daten gespeichert werden und wie auf sie zugegriffen werden kann.

In Sprachen wie RDF[3] oder OWL[4] werden Ontologien beschrieben und können von Anwendungen, die diese Sprachen unterstützen, genutzt werden. Ob die Ontologie in einer Datenbank, einer Datei oder anders gespeichert

[3] Ressource Description Framework
[4] Ontology Web Language

wird, ist dabei irrelevant. Ontologien bilden zwischen einzelnen Anwendungen im Grunde eine Netzsprache, über die kommuniziert wird. In ihnen ist definiert, wie die Daten repräsentiert werden.

In verteilten Systemen beschäftigt man sich unter anderem damit, wie Marshalling und Unmarshalling stattfindet, wie Daten über das Netz verständlich ausgetauscht werden können. Bei der Verwendung von Ontologien ist das Datenformat, die Netzsprache, von vornherein festgelegt durch die Sprache der Ontologie. Durch die Sichtweise der Ontologie entfernt man sich von der konkreten Implementierung der Datenspeicherung in einer Datenbank oder Datei und betrachtet ausschließlich die Datendarstellung abstrahiert von der Datenspeicherung.

Ontologien dienen in verschiedenen Bereichen als Mittel zur Strukturierung und zum Datenaustausch, um bereits bestehende Wissensbestände zusammenzufügen - beispielsweise genetische Daten in der Bioinformatik. Experten aus verschiedenen Gebieten müssen sich lediglich um die Modellierung ihres jeweiligen Spezialwissens und die dafür notwendigen Inferenzprozesse kümmern. Auf diese Weise können deklaratives Wissen, Problemlösungstechniken und Schlussfolgerungsmechanismen von mehreren Systemen geteilt werden.

Ontologien haben mit der Idee des *Semantic Web* in den letzten Jahren einen Aufschwung erfahren. Eine allgemeine Definition des Begriffes ist schwierig, da je nach Autor verschiedene Systeme darunter subsummiert werden. So lassen sich mehrere bereits länger existierende Formate und Ansätze wie Frames und Semantische Netze aus der Künstlichen Intelligenz oder Klassifkationen und Thesauri aus der Dokumentationswissenschaft wahlweise als Vorläufer oder als spezielle Formen von Ontologien auffassen.

Formale Sprachen zur Beschreibung von Ontologien sind unter anderem RDF-Schema, DAML+OIL, F-Logic, die vom World Wide Web Consortium für das semantische Web propagierte Ontology Web Language (OWL), die Web Service Modeling Language (WSML) und die unter ISO/ IEC 13250:2000 normierten Topic Maps. [18]

Folgende Definition lassen sich finden zu dem Begriff Ontologie:

- Eine Menge von generischen oder philosophischen Konzepten, Axiomen und Beziehungen von Domain-Ontologien.
- Eine Taxonomie von weltlichen Termen oder Kategorien einschließlich Definitionen, hierarchischen Beziehungen oder formalen Axiomen.
- Eine Menge von Klassendefinitionen und ihren Beziehungen.
- Ein Katalog von Typisierungen von Dingen organisiert durch klassenhierarchische Beziehungen.

- Schemata von Metadaten mit maschinenverarbeitbaren Semantiken.
- Inhaltliche Theorien über die Art von Dingen, deren Eigenschaften und Beziehungen.

Im einfachsten Falle versteht man unter einer Ontologie eine Klassenhierarchie im objektorientierten Sinne. Im Verlaufe der Diplomarbeit verwende ich den Begriff der Ontologie als Konzeptionierung von semantischen Zusammenhängen, die konkret in OWL[5] implementiert wurde und als eine erweiterte Klassenhierarchie angesehen werden kann mit verschiedenen, innerhalb der Ontologie existierenden Instanzen dieser Klassen.

3.8.1 Beispiel einer Ontologie

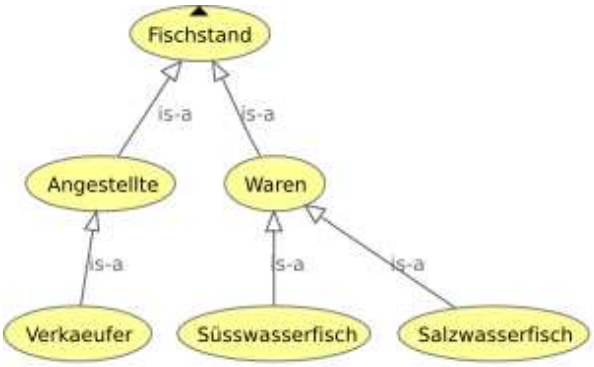

Abbildung 4: Ontologie Fischverkauf

Um das Konzept von Ontologien zu veranschaulichen, modellierte ich mit dem Ontologie-Editor Protégé[21] eine Ontologie zu einem Fischstand, der Fisch verkauft und in komplexerer Form auf jedem Wochenmarkt gefunden werden kann. Die Ontologie zu dem Fischstand visualisierte ich mit OWLViz[6], was in Abbildung 4 zu sehen ist. Die hier dargestellte Ontologie ähnelt einer Klassenhierarchie. Beziehung zwischen verschiedenen Knoten bei einer Ontologie sind natürlich auch möglich. Die hier dargestellte Beziehung *is-a* ist nicht zu verwechseln mit der gleichlautenden Beziehung in der objektorientierten Software-Entwicklung. Bei Ontologien wird von der übergeordneten Klasse nicht geerbt, sondern die übergeordnete Klasse ist lediglich semantisch verknüpft.

[5] Ontology Web Language
[6] Plugin für Protege zur Visualisierung von Ontologien

4 JADE

In Kapitel 3 wurden die Technologien beschrieben, die zur Verbesserung der in Kapitel 2 aufgezeigten Ausgangssituation beitragen können. In diesem Kapitel wird das Framework JADE beschrieben mit dem eine dieser Technologien, Agenten, implementiert werden kann. Anschließend wird eine mögliche Implementierung an einem Beispiel-Agentensystem veranschaulicht. Im Verlauf meiner Diplomarbeit verwendete ich zur Entwicklung von Multi-Agenten-Systemen das Framework JADE[7] für die Programmiersprache Java. Es vereinfacht insofern die Entwicklung, da es eine API[8] mit vordefinierten Klassen zur Verfügung stellt, um agentenspezifische Implementierungen zu ermöglichen. Außerdem stellt es eine graphische Benutzerschnittstelle (GUI) zur Verfügung, mit dem Agenten beobachtet, belauscht, gestartet, geklont und gestoppt werden können.

4.1 Was ist JADE?

JADE (**J**ava **A**gent **DE**velopment Framework) ist ein Software-Framework, was komplett in der Programmiersprache Java implementiert ist. Es vereinfacht die Implementierung von Multi-Agenten-Systemen durch eine Middleware, welche die sogenannten FIPA Spezifikationen erfüllt, und durch eine Reihe graphischer Werkzeuge, welche die Test- und Integrationsphase unterstützen. FIPA ist ein Gremium zur Spezifikation der Agenten-Kommunikation und wird später in diesem Kapitel beschrieben. Die Agentenplattform kann über mehrere Rechner verteilt werden (selbst, wenn sie nicht das gleiche Betriebssystem benutzen) und durch ein GUI überwacht werden. Die

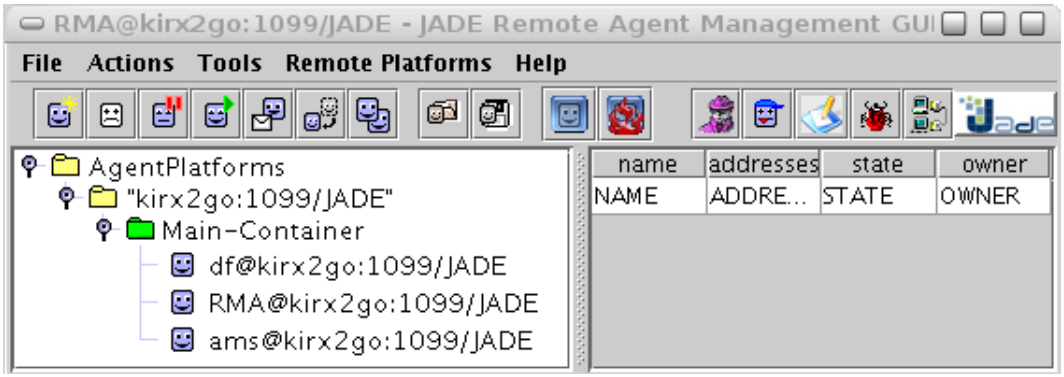

Abbildung 5: GUI mit den drei Standardagenten

[7] http://jade.tilab.com/
[8] Application Programming Interface

Konfiguration kann selbst in Laufzeit geändert werden, indem man Agenten von einem Rechner auf einen Anderen bewegt. Die von JADE benutzten Standardagenten sind, wie in Abbildung 5 zu sehen ist:

- **DF:** die Directory Facility, welche eine Art Gelbe Seiten des Systems darstellt. Hier können sich Agenten registrieren und ihre Dienste propagieren.
- **RMA:** der Remote Management Agent, welcher das graphischer Benutzerinterface handhabt.
- **AMS:** der Agent Management Service, welcher der Kernagent ist und alle JADE-Programme und Agenten überwacht. Über ihn kommunizieren die Agenten miteinander.

4.2 DF-Directory Facility/Die Gelben Seiten

Der Dienst des **DF**, der vergleichbar ist mit einer Art Gelben Seiten, erlaubt es Agenten einen oder mehrere Dienste, die sie anbieten, zu propagieren und so anderen Agenten zugänglich zu machen. Ein Dienst ist als simpler String im DF gespeichert mit der zugehörigen ID des Agenten. Agenten mit gleichem Dienst können anhand ihrer ID unterschieden werden. Bietet ein Agent mehrere Dienste an, so ist mehrmals seine ID im DF gespeichert mit unterschiedlichem String als Dienst. Andere DF-Agenten können neben dem Standard-DF-Agent ebenfalls agieren, müssen aber zusammen kooperieren, so dass ein einziger verteilter Gelbe-Seiten-Katalog verfügbar ist. Im Folgenden ist beispielhaft gezeigt, wie eine solche Registrierung in einem Agent implementiert würde.

```
ServiceDescription sd = new ServiceDescription();
sd.setType("propagierterDienst");
sd.setName(this.getName());
sd.setOwnership("Owner");
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
dfd.addServices(sd);
try {
   DFService.register(myAgent, dfd);
} catch (Exception ex) {
   System.out.println("Failed registering with DF!");
}
```

Der Agent registriert sich hier über seine Agenten-Identifizierung AID im DF mit dem von ihm angebotenen Dienst „*propagierterDienst*“. Somit können

nun andere Agenten, die nach „*propagierterDienst*“ suchen im DF seine AID erhalten und direkt mit ihm kommunizieren:

```
DFAgentDescription dfd = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("propagierterDienst");
dfd.addServices(sd);
SearchConstraints ALL = new SearchConstraints();
ALL.setMaxResults(new Long(-1));
try {
   DFAgentDescription[] result = DFService
           .search(this, dfd, ALL);
   AID[] agents = new AID[result.length];
   for (int i = 0; i < result.length; i++) {
      agents[i] = result[i].getName();
   }
} catch (FIPAException fe) {
   fe.printStackTrace();
}
```

In der Liste *agents* vom Typ AID werden alle Agenten gespeichert, die im DF registriert sind mit dem Dienst „*propagierterDienst*“.

4.3 RMA-Agent

Der **Remote Management Agent** (RMA) ist nur dafür zuständig die Benutzeroberfläche, die zur Administration genutzt wird, darzustellen und auf dem entsprechenden graphischen Interface auszugeben. So kann die GUI lokal auf dem Rechner laufen oder, wenn der Agent von einem anderen Rechner aus gestartet wird, auf diesem dargestellt werden. Mehrere Agenten laufen immer in einer Laufzeitumgebung, können aber auf mehrere Rechner verteilt werden, solange diese sich in derselben Laufzeitumgebung befinden. Ebenso kann die Benutzeroberfläche auf einem entfernten Rechner gestartet werden, solange sich der Rechner in der Laufzeitumgebung befindet.

4.4 AMS - Agent Management Service

Der **Agent Management Service** (AMS) verwaltet den Namensdienst unter JADE. So kann jeder Agent über einen eindeutigen Identifikator innerhalb der Laufzeitumgebung angesprochen werden. Dieser vom AMS verwaltete Identifikator nennt sich Agent ID und mit ihm kann ein Agent eine Nachricht eindeutig an einen anderen Agenten adressieren. Ebenso stellt der AMS die Autorität innerhalb der Laufzeitumgebung dar. Auf Anfrage von Agenten kann über die AMS ein anderer Agent, selbst wenn er sich dann in einem anderen Container befindet, erstellt oder vernichtet werden.

4.5 Introspector

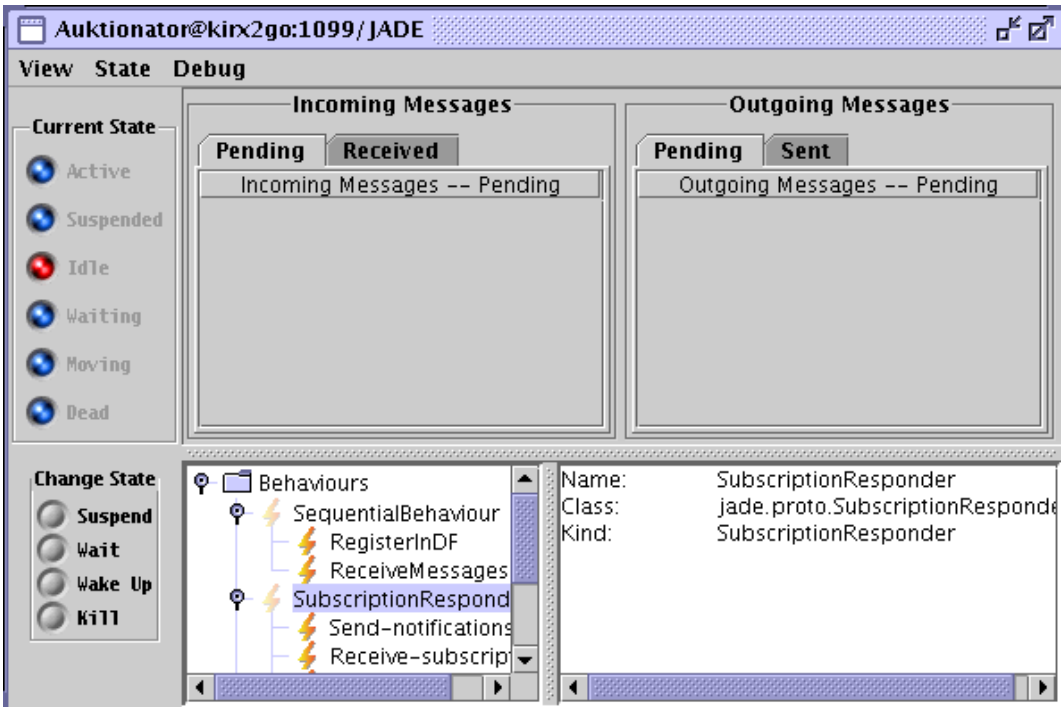

Abbildung 6: Das Monitoring-Werkzeug Introspector

JADE stellt das Monitoring-Werkzeug Introspector zur Verfügung mit dem man die Laufzeit eines einzelnen Agenten überwachen kann. Introspector bietet so zum Beispiel Möglichkeiten zum Debugging und zur Überwachung der Performance. Es kann zur Laufzeit überwacht werden, wann sich ein Agent in welchem Verhalten befindet. Es wird angezeigt, welche Verhalten gerade aktiv sind, welche Nachrichten im Ausgang und Eingang auf ihren Versand oder ihre Bearbeitung warten. Was ein Verhalten bei einem Agentensystem und insbesondere im JADE-Framework bedeutet, erläutere ich in Kapitel 4.7. Über den Introspector lässt sich der Agent nicht nur beobachten, sondern auch administrieren. Ein überwachter Agent kann vernichtet, gestoppt, gestartet und in Schlafzustand versetzt werden. Des weiteren lassen sich die in Ausführung befindlichen Verhalten beobachten. In Abbildung 6 sieht man den Agenten *Auktionator* mit dem Verhalten *SequentialBehaviour*, welches als Unterverhalten *RegisterInDF* und *ReceiveMessages* gestartet hat. Zusätzlich zum Verhalten *SequentialBehaviour* ist noch das Verhalten *SubscriptionResponder* aktiv. Da dieses angeklickt ist, wird sein Name, seine Klasse und sein Typ angezeigt.

4.6 FIPA

Die Spezifikation, die von JADE eingehalten werden, wurden von der Organisation FIPA erstellt. Die FIPA ist eine Standardisierungsorganisation und Ableger der IEEE Computergesellschaft. FIPA steht für *The **F**oundation for **I**ntelligent **P**hysical **A**gents*. Sie fördert agentenbasierte Technologien und die Interoperabiliät ihrer Standards mit anderen Technologien. Sie wurde am 8. Juni 2005 als elfte Standardisierungskomission der IEEE akzeptiert.

Die FIPA wurde ursprünglich 1996 als eine schweizerische Organisation gegründet, um Spezifikation für Softwarestandards in heterogenen und interagierenden Agentensystemen festzulegen. Seit seiner Gründung hat die FIPA eine essentielle Rolle bei der Entwicklung von Standards für Agenten gespielt und hat zahlreiche Initiativen und Events gefördert, die sich mit der Entwicklung und Erhaltung von Agenten-Technologien befasst haben. Des weiteren kommen viele der Ideen, die ursprünglich von der FIPA entwickelt wurden in das Blickfeld neuer Generationen der Netz, Internet-Technologien und verwandter Zweige. [17]

4.7 Verhalten

Ein Agent soll unabhängig agieren und parallel dazu Aufgaben ausführen, um mit anderen Agenten zu kommunizieren oder Veränderungen seiner Umwelt zu bewirken. Diese Aufgaben parallel auszuführen, ließe vermuten jedem Agenten einen Thread zuzuordnen, was in JADE auch so gemacht wird. Um weitere Parallelität innerhalb eines Agenten zu erhalten, wäre es naheliegend mehrere Threads einem Agenten zu geben. Jeder Thread erhält in den bisherigen Java-Versionen einen Betriebssystem-Thread. Der Übergang aus einem Thread in den nächsten Thread ist circa 100 Mal langsamer, als eine Methode aufzurufen. Des wegen erhält jeder Agent nur einen Thread und für weitere Nebenläufigkeit erhält er statt weiteren Threads sogenannte **Verhalten** oder **Behaviour** in Englisch.

Ein Verhalten ist ein ***Event Handler***, also eine Methode, die beschreibt, wie ein Agent auf ein Ereignis reagiert. Formell ist ein *Ereignis* die relevante Änderung eines Zustandes. Dies ist zum Beispiel der Erhalt einer Nachricht oder das Ablaufen eines Timers. In JADE sind **Verhalten** Klassen und der Code des ***Event Handlers*** des Verhaltens ist in der Methode *action* zu finden. Jedes Verhalten spiegelt die Ausführung einer **einzelnen** aktiven Phase wieder.

Ein Scheduler, der in der Basisklasse *Agent* implementiert ist und der nicht sichtbar ist für den Programmierer, plant nach nicht-preemptiven Round-Robin-Verfahren alle Verhalten in der aktuellen Schlange. Das sich in Aus-

führung befindende Verhalten kann nicht unterbrochen werden bis es die Kontrolle zurück gibt dadurch, dass die Methode *action* beendet wird. So gesehen ist ein Verhalten, was sich in Ausführung befindet, ***atomar***. [16]

4.8 Nachrichtenaustausch zwischen den Agenten

Die Agenten in JADE laufen in einer geschlossenen, logischen Laufzeitumgebung. Die Agenten sind jedoch nicht lokal beschränkt, sondern können sich auch auf mehrere Rechner verteilen. Diese Laufzeitumgebung, in der sie gestartet werden, stellt einen logischen Zusammenschluss der Agenten dar. Innerhalb dieser Laufzeitumgebung ist die Kommunikation über sogenannte **ACL**[9]-Nachrichten geregelt. Zum Austausch einer Nachricht muss dem sendenden Agenten nur die *AID*[10] des empfangenden Agenten bekannt sein. Diese *AID* erhält er zum Beispiel über die gelben Seiten, indem er nach angebotenen Services sucht. Eine solche **ACL**-Nachricht beinhaltet seine Menge von Attributen, die die FIPA-Spezifikationen erfüllen. Der Nachrichtenaustausch wird über den Agent Management Service organisiert.

Ein Agent, der eine Nachricht verschickt, erstellt konkret ein Objekt des Typs *ACLMessage*, füllt die Attribute der Nachricht entsprechend seinen Bedürfnissen aus und verschickt sie durch den Aufruf der Methode *Agent.send()*. Ähnlich verhält es sich beim Empfang von Nachrichten. Ein Agent, der Nachrichten empfangen will, ruft die Methode *Agent.receive()* auf, welche als Rückgabewert die ACL Nachricht beinhaltet. Das Empfangen und Senden von Nachrichten wird üblicherweise in einem *ReceiveBehaviour* und einem *SendBehaviour* verarbeitet.

4.9 Beispiel eines Agentensystems in JADE

Um die Funktionalität von JADE kennen zu lernen und um die grundlegenden Prinzipien eines Agentensystems zu veranschaulichen, entwickelte ich ein Beispielagentensystem über die Verwaltung und den Zugriff auf Auktionen. Dafür adaptierte ich ein bestehendes Agentensystem aus dem Primer-Tutorial von JADE, dem Bankbeispiel[20]. In diesem Beispiel werden zwei Agenten kreiert, die jeweils die Client-Rolle und Server-Rolle einer Bank mit Konten implementieren. Die *BankClientAgent*-Klasse agiert als Client und die *BankServerAgent*-Klasse agiert als Server. Beide Klassen implementieren das gemeinsame Interface *BankVocabulary*, was die Konstanten definiert, welche die Terme der gemeinsamen Sprache der Agenten repräsentiert, der Agentenontologie.

[9] Agent Communication Language
[10] Agent IDentifier

In meinem Beispielagentensystem existieren drei verschiedene Typen von Agenten:

- **Auktions-Agent**: Der Auktions-Agent spielt die Rolle des Auktionators. Er nimmt neue Auktionen auf, verwaltet das Auslaufen dieser Auktionen und nimmt Gebote auf die Auktionen an.
- **Benutzer-Agent**: Der Benutzer-Agent verwaltet Anfragen vom Benutzer, der somit neue Auktionen erstellen kann, sowie auf bereits erstellte Auktionen bieten kann.
- **Biet-Agent**: Der Biet-Agent nimmt Aufträge an und bietet autonom auf Auktionen, solange er nicht Höchstbietender ist, der Maximalbetrag des Auftrags nicht erreicht ist oder die Auktion noch nicht ausgelaufen ist.

Die Kommunikation der Agenten wird über die Agentenontologie definiert, welche die ausgetauschten Nachrichtentypen beschreibt, was sie wann für Inhalte haben und mögliche auftretende Probleme definiert.

4.9.1 Auktions-Agent

Der Auktions-Agent nimmt Nachrichten von den anderen Agenten entgegen. Diese Nachrichten beinhalten Anfragen entweder neue Auktionen zu erstellen, oder auf Auktionen zu bieten. Ausgelaufene Auktionen werden als beendet gewertet und auf sie kann nicht mehr geboten werden.

Um alle anderen Agenten auf dem Laufenden zu halten über Änderungen an den Auktionen oder über neu erstellte Auktionen, habe ich das Publisher-Subscriber-Design-Pattern[22] verwendet. Der Auktions-Agent übernimmt dabei die Rolle des Publishers und der Benutzer-Agent sowie der Biet-Agent übernehmen die Rolle des Subscribers. Sobald eine neue Auktion erstellt wurde oder auf eine Auktion geboten wurde, wird die Liste der Auktionen an alle registrierten Agenten geschickt.

Die Implementierung dieses Design-Patterns erfolgte, indem von den in der API des JADE-Framework vorgegebenen Verhalten *SubscriptionResponder* und *SubscriptionInitiator* geerbt wurde und diese erweitert wurden. Das Verhalten *SubscriptionResponder* übernimmt hierbei die Rolle des Publishers. Beim Erben musste ein sogenannter *SubscriptionManager* implementiert werden, der die registrierten *Subscriber* verwaltet, indem er neue *Subscriber* aufnimmt und *Subscriber*, die sich de-registrieren, entfernt. Ebenso werden bei einer Erstellung oder Änderung einer Auktion alle registrierten

Subscriber vom *SubscriptionManager* darüber informiert. Der *SubscriptionManager* selber ist auch wieder ein Verhalten, was ein *CyclicBehaviour* erweitert und bei jedem Durchlauf der Funktion *action* überprüft, ob eine Änderung der Auktionsliste erfolgte.

4.9.2 Benutzer-Agent

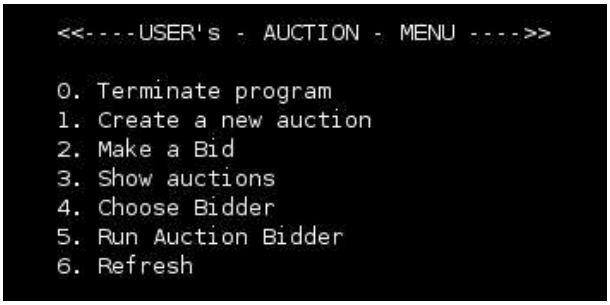

Abbildung 7: Benutzermenü

Der Benutzer-Agent stellt über die Kommandozeile ein Interface zur Kommunikation mit dem Auktions-Agenten und dem Biet-Agenten zur Verfügung. Das Interface ist in Abbildung 7 zu sehen. Der Benutzer-Agent kann neue Auktionen erstellen, auf bestehende Auktionen bieten und einen Biet-Agenten damit beauftragen, solange auf eine Auktion zu bieten bis ein Maximalbetrag erreicht wurde.

Um über Änderungen an den Auktionen, eine neue Auktion oder ein Gebot auf eine Auktion informiert zu werden, implementiert er den *Subscriber* des Publisher-Subscriber-Design-Patterns mit dem Verhalten *SubscriptionInitiatior* aus der API von JADE. Hierbei wird nur die interne Funktion *handleInform* überlagert, um die Liste mit den Auktionen bei einem *notify* vom Publisher, also dem Auktions-Agenten, zu aktualisieren.

Will der Benutzer einen Biet-Agenten mit dem Bieten auf eine Auktion beauftragen, muss er den Namen der Auktion sowie den maximal zu bietenden Betrag angeben. Anschließend kann er aus einer Liste der verfügbaren Biet-Agenten einen aussuchen, welcher dann mit dem Bieten beauftragt wird.

4.9.3 Biet-Agent

Der Biet-Agent nimmt Aufträge von Benutzer-Agenten entgegen auf Auktionen zu bieten, solange bis er entweder der Höchstbietende ist, der Maximalbetrag erreicht ist oder die Auktion abgelaufen ist. Auch er registriert sich

wie der Benutzer-Agent beim Auktions-Agenten, um über neue Auktionen und Änderungen von Auktionen auf dem Laufenden gehalten zu werden. Somit ist er eingebettet in eine gewisse Umgebung, da er auf seine Umgebung Einfluss nehmen kann, indem er über Auktionen informiert wird und auf Auktionen bieten kann. Autonom handelt er in dem Sinne, dass er vom Benutzer Aufträge, auf Auktionen zu bieten, entgegen nehmen kann und nach einem selbständigen Muster versucht, Gewinner dieser Auktion zu werden. Die konkrete Implementierung ist recht primitiv gehalten, nämlich indem er immer den Betrag plus Eins bietet, falls er nicht Höchstbietender ist oder der Maximalbetrag erreicht ist. Die Flexibilität, die von einem Agenten gefordert ist, ist bei diesem Beispiel der situative Kontext des Abgebens von Geboten auf Auktionen, wobei Gebote von anderen Agenten auf Auktionen ebenfalls zur Kenntnis genommen werden und eine reaktive Konsequenz, dem überbieten, zur Folge hat.

Ein Problem, was sich mir bei der Implementierung stellte, war, dass Auktionen nach einem Gebot nicht direkt mit dem Auktions-Agenten abgeglichen wurden und der Agent ununterbrochen auf die gleiche Auktion bot, weil er ja nicht als Höchstbietender gesetzt war in der Auktion. Durch die Implementierung des Verhaltens **Bieten** als *CyclicBehaviour* wurde die Methode *action* mehrmals aufgerufen bevor die aktuellen Auktionen vom Publisher den Biet-Agenten erreichten. Der Leerlauf des Verhaltens, was für den Nachrichtenempfang zuständig war, zeigte sich als verantwortlich für diese Problematik. Abhilfe erreichte ich durch eine Art Semaphore, die überprüft, ob auf die Auktionen innerhalb des Biet-Agenten zugegriffen werden kann oder nicht. Wurde ein Gebot auf eine Auktion gesetzt, kann kein weiteres Gebot ausgeführt werden bis nicht die Auktionen mit dem Auktions-Agenten abgeglichen wurden.

4.9.4 Agentenontologie

Eine Agentenontologie ist eine programmspezifische Ontologie, welche die Elemente beschreibt, die als Inhalt der Agentennachrichten verwendet werden können. Eine solche Ontologie besteht aus zwei Teilen: einem Vokabular, das die Terminologie der Konzepte beschreibt, die von den Agenten bei der Kommunikation verwendet werden, und die Nomenklatur der Beziehungen innerhalb dieser Konzepte, was die Semantik und Struktur darstellt. Man implementiert eine solche Ontologie, indem man die in JADE vordefinierte Klasse *Ontology* erweitert und eine Gruppe von Elementenschemata hinzufügt, welche die Struktur der Konzepte, Aktionen und Prädikate der Inhalte der Agentennachrichten beschreibt.

Bei der Definition einer Ontologie werden normalerweise die drei ver-

schiedenen Interfaces *Concept*, *AgentAction* und *Predicate* verwendet. Die korrespondierenden Klassen, welche in der Ontologie verwendet werden sind respektive *ConceptSchema*, *AgentActionSchema* und *PredicateSchema*. Neben diesen drei Interfaces mit denen man die abstrakten Objekte der Agentenontologie definieren kann, sieht JADE ebenfalls vor, atomare Elemente zu definieren, welche die Slots der abstrakten Konzepte darstellen, wie zum Beispiel *String*, *Integer*, *Float* und so weiter. Die Unterstützung dieser Art von atomaren Objekten ist durch die Klasse *PrimitiveSchema* gewährleistet. [20]

In Anwendung dieser Prinzipien entwickelte ich an die vom Bankbeispiel angelehnten Ontologieklassen, welche die in Anwendungsfällen auftretenden Gegebenheiten definieren. Bei der Kommunikation zwischen

- **Auktions-Agent**,
- **Benutzer-Agent** und
- **Biet-Agent**

muss klar sein, was für Daten übermittelt werden müssen damit die Kommunikation eindeutig und transparent ist. Als Beispiel eines eindeutigen Infor-

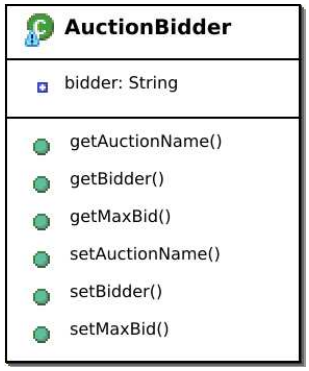

Abbildung 8: Ontologieklasse

mationsaustausches bei der Kommunikation zwischen einem Benutzer und einem Agenten stelle ich hier die Ontologieklasse *AuctionBidder* ausführlicher vor, die als UML-Diagramm in Abbildung 8 zu sehen ist. Sie ist als Ontologie beschrieben, damit ein Benutzer-Agent einen Biet-Agenten beauftragen kann, bei einem Auktions-Agenten auf eine bestimmte Auktion solange zu bieten bis das Maximalgebot erreicht ist. Notwendigerweise wird eine solche Ontologieklasse definiert, damit Agenten außerhalb der vom Programmierer vorgesehenen Programmabläufe des Agentensystems ebenfalls mit dem

Agenten kommunizieren können, solange sie die verwendeten Ontologien implementieren.

Die zur Kommunikation notwendigen Daten, die von der Ontologie *AuctionBidder* beschrieben werden, sind der Name der Auktion auf die geboten werden soll, das Höchstgebot für die Auktion auf die geboten werden soll und der Name des Auftraggebers unter dessen Namen auf die Auktionen geboten werden. In dem Vokabular, dem Interface *AuctionVocabulary*, werden die Slots definiert, die wiederum in der Klasse *AuctionBidder* als Attribute stehen müssen.

```
public interface AuctionVocabulary {
   ...
   public static final String AUCTION_BIDDER = "AuctionBidder";
   public static final String AUCTION_BIDDER_AUCTIONNAME = "auctionname";
   public static final String AUCTION_BIDDER_MAXBID = "maxbid";
   public static final String AUCTION_BIDDER_BIDDER = "bidder";
   ...
}
```

Die Strings *auctionname*, *maxbid* und *bidder*, die im Vokabular definiert werden, finden sich als Attribute in der Klasse *AuctionBidder* wieder. Für den Zugriff auf diese Attribute werden Get- und Set-Methoden implementiert, welche erforderlich sind, weil die eigentlichen Klassenattribute eines Ontologieobjekts privat deklariert sind.

```
public class AuctionBidder implements AgentAction {
   private String auctionname;
   private float maxbid;
   private String bidder;
   public void setAuctionName (String auctionname) {
      this.auctionname = auctionname;
   }
   public void setMaxBid (float maxbid) {
      this.maxbid = maxbid;
   }
   public void setBidder (String bidder) {
      this.bidder = bidder;
   }
   public String getAuctionName () {
      return this.auctionname;
   }
   public float getMaxBid () {
      return this.maxbid;
   }
   public String getBidder () {
     return this.bidder;
   }
}
```

Ansonsten kann nachher die eigentliche Ontologie, die *AuctionOntology*, nicht die definierten Ontologieklassen verwenden. Die Namenskonventionen sind konsistent zu halten. Wird im Auktionsvokabular AUCTION_BIDDER der Wert *AuctionBidder* zugewiesen, ist so ebenfalls die Klasse des Ontologieobjekts zu bezeichnen. Ebenso verhält es sich mit den Namen der Slots und den Attributen in der Ontologieklasse sowie den Get- und Set-Methoden. Hat der Slot im Vokabular AUCTION_BIDDER_AUCTIONNAME den Wert *auctionname* zugewiesen bekommen, hat das Attribut in der Ontologieklasse den Bezeichner *auctionname* zu tragen, die Get-Methode get*Auctionname* und die Set-Methode set*Auctionname* zu heißen.

Abschließend werden der eigentlichen Ontologie die Klassenontologien hinzugefügt. Zu beachten bei dieser Implementierung ist, dass nur obligatorische Slots definiert werden, die bei der Kommunikation über eine Ontologieklasse angegeben werden müssen. Wirkliche Beziehungen der Klassenontologien untereinander oder innerhalb einer Klassenontologie habe ich aus Gründen der Komplexität nicht implementiert.

```
public class AuctionOntology extends Ontology
                              implements AuctionVocabulary {
   ...
   AgentActionSchema as = new AgentActionSchema(AUCTION_BIDDER);
   add (as, AuctionBidder.class);
   as.add(AUCTION_BIDDER_AUCTIONNAME,
               (PrimitiveSchema) getSchema(BasicOntology.STRING),
               ObjectSchema.MANDATORY);
   as.add(AUCTION_BIDDER_MAXBID,
               (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
               ObjectSchema.MANDATORY);
   as.add(AUCTION_BIDDER_BIDDER,
               (PrimitiveSchema) getSchema (BasicOntology.STRING),
               ObjectSchema.MANDATORY);
   ...
}
```

Die Implementierung dieser sogenannten Agentenontologie findet in JADE als Programmierung eines eigenen Pakets in Java für die Ontologie statt, welches dann von den eigentlichen Agenten verwendet wird. Ein Tool zur Modellierung von Ontologien, wie Protégé, kann hier leider nicht angewendet werden und die Ontologie muss kodiert in Java entworfen werden. Die Klassen aus anderen Paketen benutzen dann jeweils eine Instanz der Ontologieklasse. Mit Protégé entworfene Ontologien sind entweder in einem proprietären Format „*PPJ*“ oder in einer „*OWL*“-Datei gespeichert.

4.9.5 Das Zusammenspiel der Agenten

Die Reihenfolge beim Starten der Agenten spielt insofern eine Rolle, da in dieser Beispielarchitektur die Biet- und Benutzer-Agenten den Auktions-Agenten im DF [11] auffinden müssen, um sich bei diesem als Subscriber zu registrieren. Der Auktions-Agent fragt im DF nach, ob sich ein Agent mit

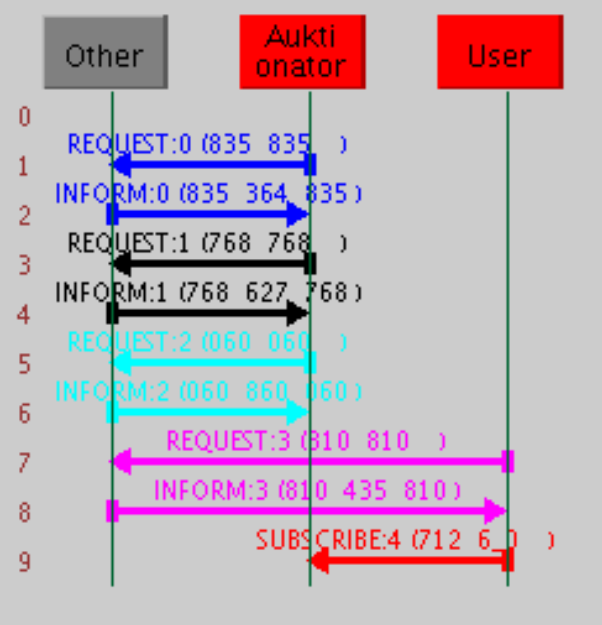

Abbildung 9: DF-Registrierung

seinem Namen registriert hat. Ist ein Agent mit seinem Namen vorhanden, wird dieser entfernt. Vorsicht ist dabei geboten, weil von einem Agenten auch andere Agenten, die im DF registriert sind, gelöscht werden können. Als Antwort erhält er die Information über eine erfolgreiche Löschung. Anschließend registriert er sich im DF. Benutzer-Agenten und Biet-Agenten können somit im DF nach einem Auktions-Agent suchen. Es können auch mehrere Auktions-Agenten im DF registriert sein. Nachdem der Benutzer- oder Biet-Agent die ID eines Auktions-Agenten erhalten hat, die Agent ID, kurz AID, kann er sich bei ihm als Subscriber registrieren, um über Änderungen der Auktionen informiert zu werden.

Zu sehen ist ein solcher Vorgang in Abbildung 9: Der Auktions-Agent Auktionator fragt nach, ob sich ein Agent mit seinem Namen schon registriert hat. Als Antwort erhält er die Information, dass ein Agent mit sei-

[11] Directory Facilitator; die gelben Seiten

nem Namen bereits vorhanden ist. Er stellt daraufhin die Anfrage diesen zu löschen. Anschließend fragt der Benutzer-Agent User im DF nach einem Auktions-Agenten, erhält dessen AID und *subscribed* sich bei diesem.

Ein Benutzer-Agent kann über die AID mit dem Auktions-Agenten Nachrichten austauschen. Er kann zum Beispiel eine Anfrage zur Erstellung einer Auktion versenden oder auf eine bestehende Auktion bieten, sofern diese Auktion nicht von ihm selber erstellt wurde. Die notwendigen Informationen zur Erstellung einer Auktion sind der Name der Auktion, das Startgebot und die Dauer der Auktion. Bei einem Gebot wird aus der Auktionsliste eine Auktion ausgewählt, die eindeutig über ihren Namen identifiziert ist, und der zu bietende Betrag an den Auktions-Agenten gesendet.

Des weiteren kann ein Benutzer-Agent einen Biet-Agenten damit beauftragen auf eine Auktion zu bieten. Hierfür sucht er im DF nach Biet-Agenten und erhält alle AIDs von verfügbaren Biet-Agenten. Nach Auswahl eines bestimmten Biet-Agenten sendet er die für einen Bietauftrag erforderlichen Informationen, den Namen der Auktion, sowie den maximal zu bietenden Betrag an den Biet-Agenten. Der Biet-Agent versucht nun der Höchstbietende zu bleiben, bis die Auktion beendet wird.

4.9.6 Beispiellauf

In diesem Kapitel beschreibe ich ein Szenario, um den Ablauf mehrerer Agenten zu veranschaulichen und die Funktionalitäten, sowie die Unterschiede der einzelnen Agenten herauszustellen. Nehmen wir an, es gibt die folgenden Agenten:

- Auktions-Agenten:
  - Auktionator
- Benutzer-Agenten:
  - Dieter
  - Detlef
  - Harry
  - Jürgen
- Biet-Agenten:
  - Bieter1
  - Bieter2

Benutzer Dieter erstellt eine Auktion Fisch, die eine Laufzeit von 5 Minuten hat. Das Startgebot setzt er auf 5 Euro. Benutzer Detlef, Harry und Jürgen werden über die neu erstellte Auktion vom Auktionator informiert und sind daran interessiert die Auktion zu gewinnen. Detlef und Harry entscheiden sich dafür jeweils einen Biet-Agenten zu beauftragen, um die Auktion sicher zu gewinnen. Detlef wählt Bieter1 aus und teilt ihm mit, dass sein Maximalgebot bei 8 Euro liegt. Prompt startet Bieter1 zu bieten und stellt beim Auktionator ein Gebot ab auf Fisch über 6 Euro. Harry ist ebenso daran interessiert den Fisch zu gewinnen und beauftragt Bieter2 damit, auf Fisch zu bieten mit einem Maximalgebot von 8 Euro. Bieter2 bietet nun umgehend auf den Fisch 1 Euro mehr. Das Gebot liegt also bei 7 Euro und Höchstbietender ist Harry. Bieter1 wird über die Änderungen, ebenso wie alle anderen, informiert, schreitet zur Tat und überbietet wiederum das Gebot um 1 Euro. Bieter1 hat somit Detlef als Höchstbietenden gesetzt mit 8 Euro. Bieter2 sieht das neue Höchstgebot, ist aber aus dem Rennen, weil das maximale Höchstgebot mit 8 Euro bereits erreicht ist. Jürgen hat die Gebote gemächlich verfolgt, sieht die Stabilisierung des Höchstgebotes, aber entscheidet sich dafür nicht zu bieten, weil für ihn seiner Meinung nach der Fisch mit 8 Euro viel zu teuer ist.

5 AgentOWL

In Kapitel 4 wurde das Framework JADE erklärt und an einem Beispiel veranschaulicht. In diesem Kapitel wird die bei der Entwicklung des Prototypens verwendete Bibliothek AgentOWL erläutert. AgentOWL wurde von Michal Laclavik[12] entwickelt und besteht aus Software-Bibliotheken zum Wissensmanagement in Multi-Agenten-Systemen. Die Bibliothek basiert auf JADE[13] und dem Jena Framework[14]. Jena wird in Kapitel 5.4 noch ausführlicher beschrieben. Die folgenden Funktionalitäten werden mit der Bibliothek AgentOWL abgedeckt:

- Eine Agenten-Knowledge-Base basierend auf OWL kann verwendet werden,
- Das Versenden von OWL Nachrichten,
- Das Empfangen von OWL Nachrichten,
  - Das Einfügen von empfangenen OWL-Nachrichten in das Modell,
- Der Empfang von Nachrichten basierend auf XMLRPC[15],
- Das Umwandeln von über XMLRPC empfangenen Nachrichten in RDF oder XML.

Die Bibliothek selber besteht aus den drei Klassen: **Memory**, **Ontology**, und **Message**.

5.1 Die Klasse Memory

Die Klasse *Memory* hat Methoden, um den Arbeitsspeicher der Agenten zu laden, abzuspeichern und zu manipulieren. Der Agentenspeicher basiert auf einer RDF- oder OWL-Ontologie. In Abbildung 10 ist das UML-Diagramm der Klasse dargestellt. Um auf diese Ontologien zugreifen zu können, wird das Jena Framework verwendet. Der Speicher kann aus einer Datei in der die Ontologie gespeichert ist, ausgelesen werden. Eine solche Datei kann zum Beispiel eine mit Protégé[21] erstellte OWL[16]-Datei sein. Es ist aber nicht zwingend erforderlich, dass der Agentenspeicher aus einer Datei geladen wird.

[12] http://ups.savba.sk/ misos/
[13] http://jade.tilab.com
[14] http://jena.sourceforge.net
[15] eXtended Markup Language Remote Procedure Call
[16] Ontology Web Language

Abbildung 10: Die Klasse Memory

Ebenso sind sämtliche Funktionen, wie Laden, Speichern und Manipulieren einer Ontologie mit einer Datenbank möglich. Falls eine Ontologie verwendet wird, die in einer Datenbank gespeichert ist, kann ein Agent sich im Netz bewegen und trägt den Arbeitsspeicher weiter konsistent mit sich. Er muss sich nur von der Datenbank kurzzeitig trennen, wenn er sich im Netz bewegt. Sobald er seinen neuen Standort hat, verbindet er sich erneut mit der Datenbank.

5.2 Die Klasse Ontology

Die Klasse *Ontology* hat grundlegende Konstanten zur Beschreibung von Ontologien in OWL. Diese Konstanten sind unter anderem Einträge der verwendeten Namespaces, die in jedem OWL-Dokument verwendet werden: Die Namespaces für RDF, RDFS und OWL. Des weiteren finden sich hier als Konstanten Abkürzungen wieder, die die Formulierung von RDQL-Queries vereinfachen. In der *USING*-Klausel einer RDQL-Query kann man Abkürzungen definieren, die im eigentlichen *SELECT-WHERE*-Statement verwendet werden. Eine solche Abkürzung ist der Namespace der verwendeten Ontology, der aus einer OWL-Datei ausgelesen wird und in einer Variablen gespeichert

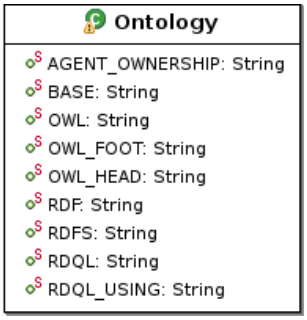

Abbildung 11: Die Klasse Ontology

wird. Somit braucht der Programmierer diesen Namespace nicht selber zu suchen und zu definieren. Die Klasse *Memory* ist als UML-Diagramm in Abbildung 11 zu sehen.

5.3 Die Klasse Message

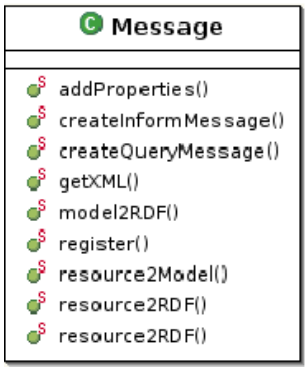

Abbildung 12: Die Klasse Message

Mit der Klasse *Message* können *Inform*- und *Query*-Nachrichten formuliert werden, die Informationen in Stringform enthalten, um von einem Agenten an einen anderen Agenten verschickt zu werden. Das UML-Diagramm der Klasse ist in Abbildung 12 zu sehen.

Ebenso können Ressourcen aus dem aktuellen Model in XML umgewandelt werden. Zusätzlich hat die Klasse Memory noch Methoden, um die einzelnen Datentypen, die man beim Zugriff auf Ontologien mit dem Jena-Framework erhält, in andere Datentypen umzuwandeln.

5.4 Das Jena-Framework

Das Jena Framework wird von der JADE-Bibliothek AgentOWL zum Zugriff, der Speicherung und dem Laden von Ontologien verwendet. Es stellt eine umfangreiche API zur Entwicklung von Semantic-Web-Applikationen zur Verfügung und beinhaltet ebenso einen Reasoner, eine regelbasierte Inferenzmaschine. Jena ist ein Open-Source-Projekt und entstand aus der Arbeit der Hewlett-Packard-Semantic-Web-Programme. Die Entwicklung des Projektes läuft unter Sourceforge und ist als solches unter http://jena.sourceforge.net zu erreichen.

Das Jena Framework beinhaltet:

- eine RDF API,
- Methoden zum Lesen und Schreiben von RDF in RDF/ XML, N3 und N-Tripels,
- eine OWL API,
- Methoden zur persistenten Speicherung von Daten im Hauptspeicher oder auf Festplatten,
- RDQL, eine Query Language zum Zugriff auf Ontologien in RDF oder OWL.

Der typische Zugriff mit einer RDQL-Query auf eine OWL-Ontologie sieht beispielhaft so aus:

```
Model m = Memory.getModel();
Query q = new Query(myRDQL_Query);
q.setSource(m);
QueryExecution qe = new QueryEngine(q);
QueryResults res = qe.exec();
for (Iterator iter = res; iter.hasNext();) {
    ResultBinding resB = (ResultBinding)iter.next();
    Resource x = (Resource)resB.get("x") ;
    // ...
    // Verarbeitung der einzelnen Resource
    // ...
}
res.close() ;
```

5.5 Demo von AgentOWL

Bei der Entwicklung meines Prototypen setzte ich auf der mitgelieferten Demoversion von AgentOWL auf und erweiterte diese Demo um die benötigten

Funktionalitäten. Die Demoversion verwendet eine simple Ontologie auf der zwei Agenten arbeiten:

- AskAgent,
- AnswerAgent.

Der AskAgent stellt Anfragen und der AnswerAgent beantwortet sie. Beide

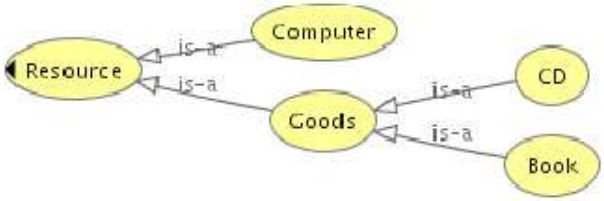

Abbildung 13: Ontologie

Agenten greifen auf dieselbe Ontologie zu, die in Abbildung 13 zu sehen ist. Jedoch hat der AskAgent keinerlei Individuen im Gegensatz zum AnswerAgent. Somit kann man schon erahnen, dass der AskAgent den AnswerAgent über vorhandene Individuen innerhalb ihrer Ontologie abfragt und der AnswerAgent ihm eine Antwort darauf liefert.

5.5.1 Benutzer-Interface

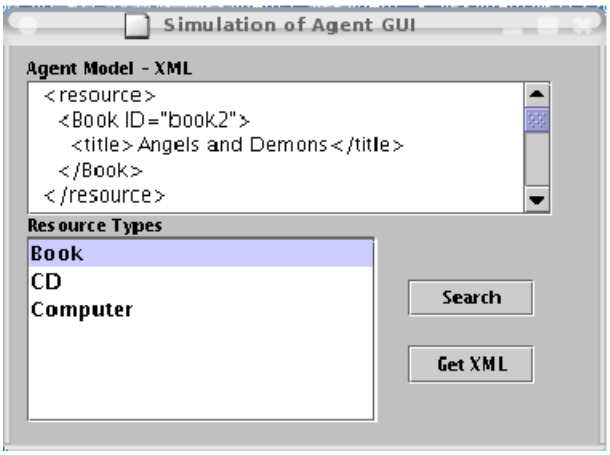

Abbildung 14: Benutzer-Interface

Zusätzlich zu den beiden Agenten AskAgent und AnswerAgent existiert noch ein Benutzer-Interface, in dem die Ressource ausgesucht werden kann, nach welcher der AskAgent den AnswerAgent fragt. Das Benutzer-Interface ist ein schlichtes Frame mit einem Textfenster, in dem der XML-Output der Antwort vom AnswerAgent ausgegeben werden kann, und einem Auswahlfenster, wo die jeweils eine der drei verfügbaren Ressourcen ausgewählt werden kann. Zu sehen ist das Benutzer-Interface in Abbildung 14.

Vom Benutzer-Interface aus können zwei Funktionen gestartet werden:

- **search**: Der Benutzer wählt eine der drei zur Verfügung stehenden Resourcen aus und das Benutzer-Interface startet über XMLRPC die Methoden Suchen im AskAgent. Dabei wird natürlich der Resourcentyp mit übergeben.
- **getXML**: Nachdem gesucht wurde, kann sich der Benutzer die Ergebnisse der Suche in XML darstellen lassen.

Diese beiden Methoden dienen zur Kommunikation mit dem AskAgent. Mit dem AnswerAgent agiert das Benutzer-Interface gar nicht.

5.5.2 Der AskAgent

Der AskAgent ist dazu da, den AnswerAgent nach der vom Benutzer-Interface übermittelten Ressource zu fragen. Der AskAgent startet zunächst einen XMLRPC-Server, um Anfragen vom Benutzer-Interface entgegen nehmen zu können. Bevor der AskAgent jedoch vollständig einsatzbereit ist, lädt er die Konfiguration der Ontologie und des Agentenspeichers, sowie aus einer OWL-Datei seine Ontologie, auf der er arbeitet:

```
AskAgent - XMLRPC Running ....
Memory Config - Trying to initiate config
Memory Config - Trying to initiate config constants
Memory Config - property file loaded from config/AskAgent.properties
AskAgent Memory - Starting with MEM model ...
AskAgent Memory - reading model from memory_init/ontology.owl
AskAgent Memory - reading individuals from memory_init/askAgent.owl
```

Nach dem Start des AskAgent registriert sich dieser im DF[17] über die bibliothekseigene Klasse *Message*:

```
Message.register(this,"AskAgent");
```

[17] Directory Facilitator - Die gelben Seiten

Wird vom Benutzer-Interface über XMLRPC die Methode *search* mit dem Übergabeparameter des Resourcentyp aufgerufen, wird vom AskAgent ein Verhalten gestartet, in dem die RDQL-Query formuliert wird, passend zum entsprechenden Ressourcentypen und eine Nachricht mit der RDQL-Query als deren Inhalt an den AnswerAgent geschickt wird. Beim Verschicken der Nachricht wird auf die bibliothekseigene Klasse *Message* zugegriffen:

```
send (Message.createQueryMessage(
        a,
        "AnswerAgent",
        "SELECT ?x WHERE (?x rdf:type ont:"+resourceType+")"));
```

Als Letztes startet er ein Verhalten, um Nachrichten vom AnswerAgent empfangen zu können. Nach einer vom AskAgent verschickten Anfrage zu Informationen über Individuen einer bestimmten Ressource, erhält er vom AnswerAgent in RDF-Format die Ergebnisse zugeschickt. Mit dem Verhalten *BehaviourHandleReceivedMessages* empfängt der AskAgent die Nachrichten mit den RDF-Informationen und bereitet sie für den XML-Output im Benutzer-Interface auf. Die empfangenen Ressourcen werden in

```
private Ressource agentIndividual
```

gespeichert, worauf nachher die Methode zur Umwandlung vom RDF- in das XML-Format zugreift:

```
class BehaviourHandleReceivedMessages extends CyclicBehaviour {
    public BehaviourHandleRecivedMessages(Agent _a) {
        super(_a);
    }
    public void action() {
            ...
            ACLMessage msg = receive();        ...
            OntModel m = ModelFactory.createOntologyModel();
            ...
            mem.getModel().read(new StringReader(
                                msg.getContent()),
                                Ontology.BASE);
            m.read(new StringReader(
                                msg.getContent()),
                                Ontology.BASE);
            ...
            Iterator iter = m.listSubjectsWithProperty(
                                mem.createProperty("title"));
            while (iter.hasNext()) {
                Resource r = (Resource) iter.next();
            agentIndividual.addProperty(
            mem.createProperty("resource"),
```

```
            mem.getModel().getResource(r.getURI()));
            ...
    }
}
```

Problematisch ist jedoch vom Autor der folgende Eintrag gewesen, als ich die Demo-Version erweitert habe und versucht habe, das Agentensystem auf anderen Ontologien anzuwenden:

```
Iterator iter = m.listSubjectsWithProperty(
                    mem.createProperty("title"));
```

Es werden nur zurück gegebene Ressourcen verarbeitet, die den Eintrag "**title**" als *Property* haben. Somit ist die hier implementierte Demoversion proprietär auf die vom Autor verwendete Ontologie zugeschnitten.

Die Methode *getXML*, die durch XMLRPC vom Benutzer-Interface im AskAgent aufgerufen wird, wandelt die im Speicher abgelegten Informationen in XML um und speichert sie in einem String, was auch der Rückgabeparameter der Funktion ist:

```
public String getXML(String id) {
    ...
    Resource r = mem.getModel().getResource(Memory.getBase()+id);
    ...
    xml = Message.getXML(r,"", "");
    return xml;
}
```

5.5.3 Der AnswerAgent

Der AnswerAgent nimmt Nachrichten mit RDQL-Queries als Inhalt entgegen, fragt seine Ontologie nach Instanzen von Resourcen ab und schickt die Resultate in RDF-Format als Nachricht an den AskAgent zurück, der diese dann weiter verarbeitet.

Beim Starten lädt der AnswerAgent, ebenso wie der AskAgent, die Konfiguration der Ontologie und des Speichers. Anschließend lädt er aus einer OWL-Datei seine Ontologie:

```
Ontology Config - Trying to initiate config
Ontology Config - Trying to initiate config constants
Ontology Config - Trying to initiate config constants
Ontology Config - property file loaded from Config.properties
Memory Config - Trying to initiate config
Memory Config - Trying to initiate config constants
Memory Config - property file loaded from config/AnswerAgent.properties
AnswerAgent Memory - Starting with MEM model ...
AnswerAgent Memory - reading model from memory_init/ontology.owl
AnswerAgent Memory - reading individuals from memory_init/answerAgent.owl
```

Nachdem die Konfigurationen, der Speicher und die Ontologie geladen sind, registriert sich der AnswerAgent im DF mittels der bibliothekseigenen Klasse *Message*:

```
Message.register(this,"AnswerAgent");
```

Nach der Registrierung startet der AnswerAgent ein Verhalten mit dem Nachrichten vom AskAgent entgegen genommen und verarbeitet werden können. Erhält der AnswerAgent vom AskAgent eine Nachricht mit einer RDQL-Query als Inhalt, startet das Verhalten den Jena-eigenen Reasoner, der regelbasierten Inferenzmaschine von Jena, und führt eine Abfrage gegen die Ontologie mit der RDQL-Query aus:

```
Model m = mem.getModel();
Query q = new Query(msg.getContent());
q.setSource(m);
QueryExecution qe = new QueryEngine(q);
QueryResults res = qe.exec();
for (Iterator iter = res; iter.hasNext();) {
    ResultBinding resB = (ResultBinding)iter.next();
    Resource x = (Resource)resB.get("x");
    informAskAgent(x);
 }
res.close();
```

Mit der Methode *informAskAgent*, die hier verwendet wird, startet der AnswerAgent ein Verhalten, was einmalig ausgeführt wird, um dem AskAgent die Nachricht mit dem Resultat der Query zu kommen zu lassen.

6 Die Entwicklung des Prototypen

In Kapitel 4 wurde das Framework JADE erläutert, das bei der Entwicklung zum Einsatz kam. In Kapitel 5 wurde die bei der Entwicklung verwendete Bibliothek AgentOWL für JADE dargestellt.

Dieses Kapitel stellt die Entwicklung des Prototypen zur ontologiebasierten Suche in einem MAS vor. Zur Implementierung des Agentensystems verwendete ich JADE[18]. Um auf in OWL[19] beschriebene Ontologien mit den Agenten zugreifen zu können, verwendete ich AgentOWL [23]. AgentOWL selber verwendet das Jena Framework, um Ontologien in RDF, RDFS oder OWL zu verarbeiten.

Bei der Entwicklung des Prototypen setzte ich auf die Demo von AgentOWL auf, passte sie so an, dass beliebige Ontologien verarbeitet werden können und implementierte Suchfunktionalitäten aus der von Herrn Pour-Heidari entwickelten Suchmaschine[24]. Herr Pour-Heidari verwendete als Framework zur Entwicklung dieser Suchmaschine Jena. In seiner Arbeit findet man eine ausführliche Beschreibung der Funktionalitäten dieses Frameworks. Da AgentOWL ebenso Jena verwendet, konnten die Suchfunktionalitäten auf die Agenten nach entsprechender Anpassung übertragen werden.

6.1 Austausch der Ontologien in der Demo von AgentOWL

Auf die von der Demo von AgentOWL verwendete Ontologie sind die beiden Agenten AskAgent und AnswerAgent angepasst und ein Austausch ist nicht ohne weiteres möglich. Zunächst entwickelte ich eine sehr simple Ontologie eines Fischstandes, der Fisch verkauft, wie er auf jedem Wochenmarkt in komplexerer Form vorzufinden ist. Zur Modellierung der Ontologie verwendete ich Protégé. Anschließend passte ich die beiden Agenten so an, dass sie auf dieser Ontologie arbeiten konnten.

6.1.1 Ontologie Fischverkauf

Die mit OntoViz von Protégé visualisierte Form der Ontologie ist in Abbildung 15 zu sehen. Die Ontologie Fischverkauf hat als Wurzel Fischstand. An den Fischstand angeknüpft sind die Knoten Waren und Angestellte. Als Angestellte existiert die Klasse Verkäufer und als Waren existieren Süßwasserfische sowie Salzwasserfische. Das ist die gemeinsame Ontologie, die ich für

[18] http://jade.tilab.com
[19] Ontology Web Language

Abbildung 15: Ontologie Fischverkauf

AskAgent und AnswerAgent entwickelte. Der AskAgent hat keinerlei Instanzen in der Ontologie. Der AnswerAgent hat in seiner Ontologie die folgenden Instanzen:

- Waren
  - Süßwasserfisch
    - Instanz: **Lachs**
  - Instanz: **Aal**
- Angestellte
  - Instanz: **Chef**
  - Verkäufer
    - Instanz: **Harry**

Damit die Agenten auf dieser von mir entwickelten Ontologie arbeiten können, sind in der Demo-Version die jeweiligen OWL-Dateien in dem Verzeichnis

```
./memory_init/
```

auszutauschen, das sich im Working-Directory der Installation befindet.

6.1.2 Anpassen der Agenten der Demo von AgentOWL

Startet man die Agenten mit den ausgetauschten OWL-Dateien, arbeitet der Agentenspeicher in seiner neuen Ontologie. Damit die Abfragen vom AskAgent an den AnswerAgent funktionieren, muss noch im AskAgent das Verhalten zum Empfang der Nachrichten angepasst werden. Im Kapitel 5.5.2 erläuterte ich bereits die Problematik der proprietären Implementierung des Umwandelns der vom AnswerAgent empfangenen RDF-Ressourcen zum Ablegen im Speicher des AskAgent. Mit der Implementierung der Methode

```
Model.listSubjectsWithProperty (Property p)
```

des Jena Frameworks werden aus dem *OntModel* nur die Einträge heraus gelesen, die als *Property* den Wert „**title**“ haben und somit auf die spezielle Ontologie des Verfassers angepasst sind:

```
Iterator iter = m.listSubjectsWithProperty(
                    mem.createProperty("title"));
```

Durch Ändern der Methode, damit die Ontologien im AskAgent austauschbar sind, in

```
Model.listSubjects();
```

werden schließlich alle Ressourcen aus der Nachricht vom AnswerAgent gelistet und im Speicher des AskAgent hinterlegt:

```
Iterator iter = m.listSubjects();
```

Dies führt jedoch zu der Problematik, dass Ressourcen im Speicher mehrfach hinterlegt sind. Um die doppelt vorkommenden Einträge herrauszufiltern, implementierte ich noch eine Funktionalität in der Methode *getXML*, die doppelte Zeilen aus dem XML-Output entfernte.

```
StringTokenizer st = new StringTokenizer(xml, "\n");
String a = "", b = "";
xml = "";
while (st.hasMoreTokens()) {
    a = b;
    b = st.nextToken();
    if (!a.equalsIgnoreCase(b))
    xml += b + "\n";
}
```

Nach diesen Änderungen funktionieren der AskAgent und der AnswerAgent soweit mit den ausgetauschten Ontologien. Nun musste das Benutzer-Interface an die geänderten Ressourcen angepasst werden, damit nach vorhandenen Instanzen von Ressourcen gesucht werden kann. Hierfür sind lediglich die alten Namen der Ressourcen

```
Object d[] = new Object[3];
d[0] = "Book";
d[1] = "CD";
d[2] = "Computer";
```

auszutauschen gegen diejenigen, welche sich in der Ontologie befinden:

```
Object d[] = new Object[2];
d[0] = "Salzwasserfisch";
d[1] = "Suesswasserfisch";
```

Nach diesen Änderungen ist es möglich, das Agentensystem der Demo auf beliebigen Ontologien arbeiten zu lassen. Zwischen den Agenten können Nachrichten ausgetauscht werden, welche OWL-Ontologien beinhalten, und es können Queries über die Jena-Inferenzmaschine gemacht werden. Im Folgenden beschreibe ich die von Herrn Pour-Heidari entwickelte Ontologie, die auch in meiner Diplomarbeit zum Einsatz kam.

6.2 Ontologische Basis

Auf die Entwicklung einer allgemeinen Suchontologie wurde aus Zeitgründen verzichtet und es wurde exemplarisch eine kleine Ontologie entwickelt, auf der gesucht werden kann, und die die grundlegenden Eigenschaften einer solchen Ontologie herausstellt.

Die Themengebiete, welche die Ontologie repräsentiert, sind Angebote der Stadtverwaltung für Bürger. Um die Thematiken zu katalogisieren, wurden diese als Services modelliert; Services, welche die Stadtverwaltung anbietet. Grob zu unterteilen ist die Ontologie in die Bereiche der Service-Ontologie und der Such-Ontologie.

Die Ontologie besteht im wesentlichen aus drei Klassen:

- **InformationsObjekte**: stellt die *Service-Ontologie* dar. Hierzu gehören alle Instanzen zu denen Informationen verwaltet werden.
- **Begriffssammlung**: stellt die *Such-Ontologie* dar und ist wiederum unterteilt in die Klassen *IdentifizierendeBegriffe* und *Umschreibende-Begriffe*
- **Information**: dient der Darstellung von Ressourcen der Ontologie.

6.2.1 Die Service-Ontologie

Eine mit Protégé visualisierte Darstellung der Service-Ontologie ist in Abbildung 16 zu sehen. Ein Dienstleister „erbringtDienstleistung“ und eine Dienstleistung „wirdErbrachtVon“ einem Dienstleister. Orange dargestellt sind in

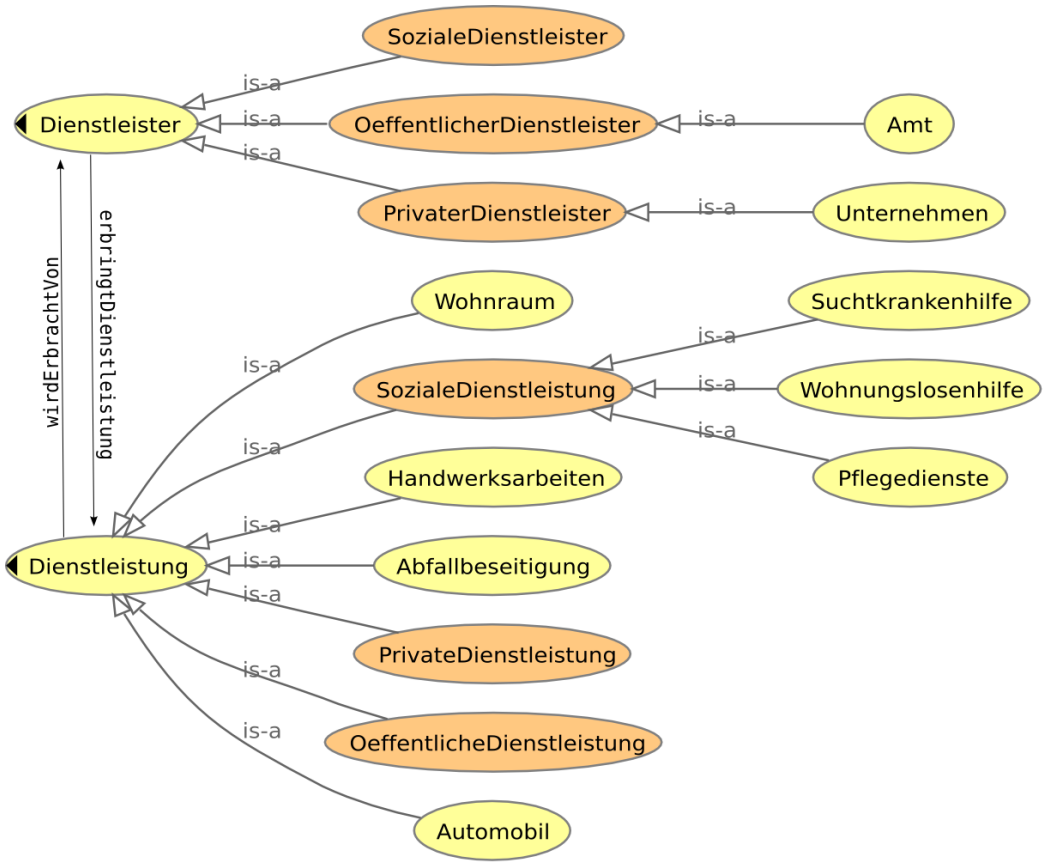

Abbildung 16: Klassen der Service-Ontologie

dieser Abbildungen die Dienstleister und Dienstleistungen bei denen diese Relation untereinander existiert.

Die Service-Ontologie ist ein Teilbereich der Ontologie und stellt die Angebote der Stadtverwaltung dar, die mit dieser Ontologie in einen konzeptionierten Zusammenhang gebracht werden können.

- **Dienstleister**: Personen, Firmen oder Institutionen, die eine Dienstleistung erbringen,
- **Dienstleistung**: Dienstleistungen jeglicher Art, wobei auch ein entsprechender Dienstleister zur Zeit nicht verfügbar sein kann.

Die Bereiche **Dienstleister** und **Dienstleistung** sind wiederum unterteilt in die folgenden drei Bereiche, wobei ein Dienstleister eine *Dienstleistung erbringt* und eine Dienstleistung von einem *Dienstleister erbracht wird*.

- **Öffentliche Dienstleister**: Repräsentierung von Ämtern und Institutionen des öffentlichen Dienstes,
- **Private Dienstleister**: Zusammenfassung von Firmen oder Einzelpersonen,
- **Soziale Dienstleister**: Anbieter von Hilfs-, Beratungs- oder Pflegediensten.

Zu beachten hierbei ist, dass einzelne Klassen sich nicht gegenseitig ausschließen müssen und Instanzen dieser Klassen mehrfach innerhalb der Ontologie auftreten dürfen. Redundanz ist also nicht ausgeschlossen.

6.2.2 Die Such-Ontologie

Ebenso wie die Service-Ontologie ist auch die Such-Ontologie ein Teilbereich der gesamten Ontologie. Beide zusammen machen die verwendete Gesamt-Ontologie aus. In der Such-Ontologie sind die Suchbegriffe in einem Suchvokabular definiert. Das Vokabular ist so gehalten, dass durch Benutzeranfragen auch Begriffe gefunden werden können, durch deren Umschreibung. Voraussetzung sind an die zu findenden Begriffe angepasste Vokabulare, die bei der Erstellung der Ontologie eingepflegt werden müssen und von denen auch die Trefferwahrscheinlichkeit abhängt. Die Vokabulare werden im Folgenden als Begriffsammlungen bezeichnet. Man unterscheidet die Definition von direkten und indirekten Begriffsammlungen, die den einzelnen Themengebieten zugeordnet sind.

Unter einer direkten Zuordnung versteht man eine Begriffsammlung, die im direkten Zusammenhang mit dem entsprechenden Inhalt steht. Es werden also außer direkten Bezeichnern noch semantisch verwandte Begriffe mit in das Vokabular aufgenommen, um eine größere Abdeckung des Spektrums zu definieren, was die Vokabulare identifiziert. Direkte Vokabulare sind in der verwendeten Ontologie als identifizierende Vokabulare in der Klasse *IdentifizierendeBegriffe* gesammelt.

Unter einer indirekten Zuordnung versteht man eine Begriffsammlung, die zwar thematisch in das Themengebiet des Dokumentes passt, jedoch nicht bezeichnend für deren Inhalt ist. Sie sollen helfen einen **thematischen Anker** zur Sucheingabe zu finden. Indirekte Vokabulare sind in der verwendeten Ontologie als beschreibendes Vokabular in der Klasse *UmschreibendeBegriffe* gesammelt.

6.2.3 Relationen innerhalb der Ontologie

Relationen innerhalb der Ontologie werden durch sogenannte Properties definiert. Diese Properties innerhalb der Ontologie sind in Abbildung 17 zu sehen. Bei den Relationen in der Ontologie werden zwei verschiedene Properties unterschieden:

- **CustomProperties**: Hier finden sich alle Relationen die Bestandteil der Ontologie sind wieder und werden durch die Service-Ontologie verwaltet. Konkret sind das die Relationen der Service-Ontologie *wirdErbrachtVon* und *erbringtDienstleistung*.
- **InformationProperties**: Das sind Relationen der Such-Ontologie, die also die Beziehungen zwischen Begriffssammlungen und Informationsobjekten beschreiben.

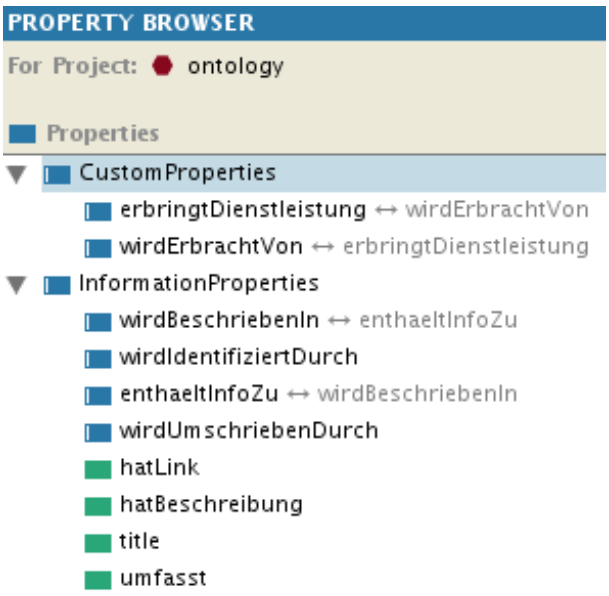

Abbildung 17: Properties der Ontologie

Innerhalb des Properties *InformationProperties* gibt es wiederum weitere acht Properties. Drei von ihnen sind lediglich dazu da, gefundene Informationen formatiert ausgeben zu können mit entsprechenden Einträgen zu den gefundenen Informationen. Hierfür wurde eigens die Klasse *Information* innerhalb der Ontologie festgelegt. Die Klasse Information hat die drei Relationen

- **hatLink**: Beinhaltet eine URL, also eine Webadresse zu der entsprechenden Information,
- **hatTitel**: Ist ein Titel der entsprechenden Information,
- **hatBeschreibung**: Beinhaltet weitere Angaben zu der entsprechenden Information.

Die vierte verwendete Relation „*umfasst*“ weist den Begriffsammlungen aus der Such-Ontologie einzelne Begriffe zu. Diese ersten vier Relationen sind in Abbildung 17 grün dargestellt.

Die restlichen vier Relationen verbinden die Service-Ontologie mit der Such-Ontologie. Dabei werden die Instanzen der Klassen der Such-Ontologie mit Instanzen der Klassen der Service-Ontologie verknüpft:

- **wirdUmschriebenDurch**: Verbindet die Klasse *Informationsobjekte* mit der Begriffssammlung *UmschreibendeBegriffe*.
- **wirdIdentifiziertDurch**: Verbindet die Klasse *Informationsobjekte* mit der Begriffssammlungg *IdentifizierendeBegriffe*.
- **wirdBeschriebenIn**: Verbindet die Klasse *Informationsobjekte* mit der Klasse *Information*.
- **enthaeltInfoZu**: Verbindet die Klasse *Information* mit der Klasse *Informationsobjekte*.

6.3 Änderung der Demo von AgentOWL

Basierend auf der angepassten Demo-Version von AgentOWL implementierte ich ein Benutzer-Interface und zwei verschiedene Agenten, um Suchfunktionalitäten in einem MAS zu gewährleisten:

- **Benutzer-Interface**: Über das Benutzer-Interface können Suchbegriffe eingegeben werden, die an den Broker-Agent delegiert werden. Die Ergebnisse werden in Triplet-Format im Benutzer-Interface dargestellt.
- **Broker-Agent**: Empfängt Anfragen vom Benutzer-Interface, formuliert daraus die RDQL-Queries und sendet diese an die Query-Agenten.
- **Query-Agent**: Nimmt die RDQL-Queries von Broker-Agenten entgegen und fragt mit ihnen den Reasoner ab. Die Resultate werden zurück an den Broker-Agent geschickt.

6.4 Das Benutzer-Interface

In diesem Kapitel beschreibe ich die für den Prototypen relevanten Änderungen der Implementierung des Benutzer-Interfaces aus der Demoversion von AgentOWL, ebenso wie relevante Funktionalitäten dieses Benutzer-Interfaces innerhalb des Agentensystems.

Das Benutzerinterface wurde soweit geändert, dass Suchbegriffe eingegeben werden können, die an einen Broker-Agent weitergeleitet werden. Das Benutzer-Interface ist mit einer beispielhaften Suchanfrage und deren Resultaten in Abbildung 18 dargestellt. Um die Resultate anzuzeigen, implementierte ich zusätzlich die Option *getResult*, welche die Resultate nicht in XML, was teilweise sehr unübersichtlich wirkt, sondern in einer Triplet-Form ausgibt:

- **Titel**: Der Titel der gefunden Ressource,
- **Beschreibung**: Die Beschreibung zur gefundenen Ressource,
- **Link**: Der Weblink zur gefundenen Ressource.

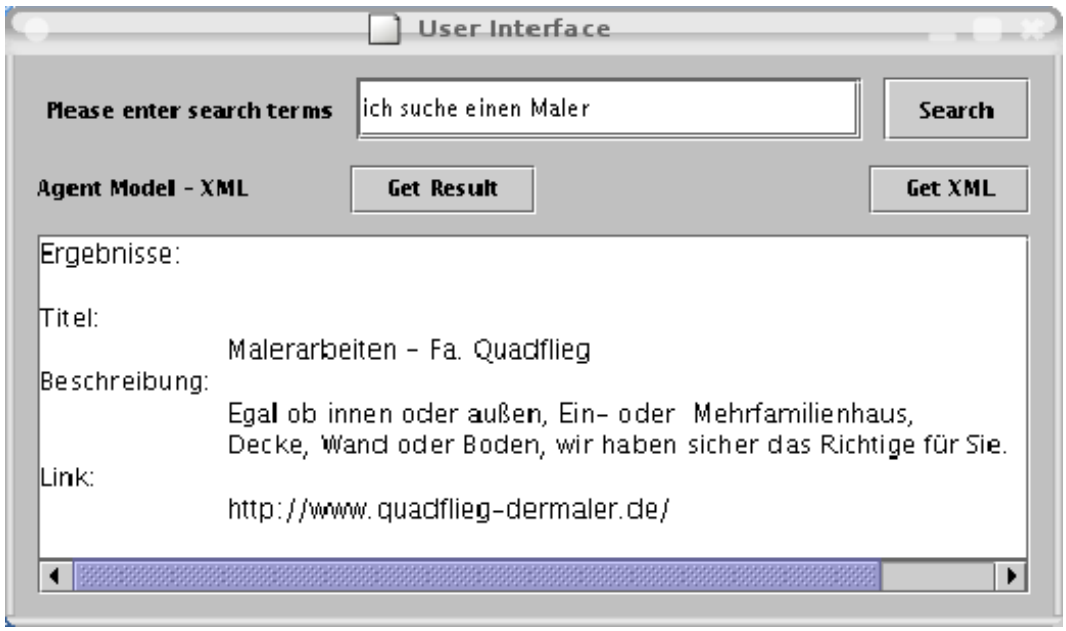

Abbildung 18: Benutzer-Interface

Um das Resultat einer erfolgten Suchanfrage zu erhalten, ruft das Benutzer-Interface über *XMLRPC* die entsprechende Funktion im *Broker-Agent* auf. Das Benutzer-Interface dient der grafischen Darstellung der implementierten

Funktionalitäten. Suchanfragen können manuell eingegeben werden in einer *TextBox* und werden an den *Broker-Agent* delegiert. Ebenso werden die Resultate im *Broker-Agent* verarbeitet und im Benutzer-Interface wird lediglich der Rückgabewerte der aufbereiteten Resultate dargestellt.

Im folgenden Kapitel werde ich auf die Funktionalitäten und die Implementierung der Suchfunktionalitäten im *Broker-Agent* eingehen.

6.5 Broker-Agent

In diesem Kapitel beschreibe ich die für den Prototypen relevanten Änderungen der Implementierung vom AskAgent aus der Demoversion von Agent-OWL, ebenso wie relevante Funktionalitäten dieses Agenten innerhalb des Agentensystems.

Beim Starten des Broker-Agents wird die Konfiguration, sowie die Ontologie in den Speicher des Agenten geladen. Die Ontologie des Broker-Agents hat keinerlei Instanzen. Erst nachdem ein Query-Agent befragt wurde, erhält der Broker-Agent Instanzen von Klassen innerhalb der Ontologie. Somit muss die Ontologie bei jeder Suchanfrage neu geladen werden, damit die verwendete Ontologie keine Instanzen im Speicher hat, denn als Rückgabe vom Broker-Agent zum Benutzer-Interface werden alle Instanzen aus der im Broker-Agent gespeicherten Ontologie übermittelt.

Die im Broker-Agent implementierten Methoden zur Generierung von Queries sind an den von Herrn Pour-Heidari entwickelten Prototypen zur ontologiebasierten Suche angelehnt und an die Gegebenheiten eines Agentensystems angepasst worden. Die originalen Implementierungen sind in seiner Arbeit zu finden [24].

Der Broker-Agent bekommt vom Benutzer-Interface die Suchwörter übermittelt. Daraus generiert er eine RDQL-Query und schickt diese an Query-Agent weiter. Anschließend nimmt der Broker-Agent die von Query-Agent ermittelten Ergebnisse entgegen und bereitet sie für das Benutzer-Interface auf.

Wird vom Benutzer-Interface ein Such-String eingegeben und anschließend ein Suchauftrag aufgegeben, wird vom Benutzer-Interface über *XML-RPC* die Methode **search** im Broker-Agent aufgerufen und als Parameter der Such-String übergeben. In der Methode **search** wird zunächst der aktuelle Speicher gelöscht und ein frisches, also leeres, Ontologiemodell geladen. Daraufhin werden alle Wörter, die sich im Such-String befinden, in einem Vektor gespeichert. Mit diesem Vektor mit den Suchwörtern als Parameter wird **rekSearch** aufgerufen, einmal mit *true* und einmal mit *false* als zweiten Parameter.

```
public void search(String searchString) {
    Vector query = new Vector ();
    mem = new Memory("config/AskAgent.properties", "AskAgent");
    agentIndividual = mem.getModel().
            getResource(Memory.getBase() + "AskAgent");
    StringTokenizer st = new StringTokenizer(searchString, " ");
    while (st.hasMoreTokens()) {
        query.add(st.nextElement());
    }
    rekSearch(query, true);
    rekSearch(query, false);
    return;
}
```

In der Methode **rekSearch** werden die als Vektor übergebenen Suchwörter permutiert. Rekursiv wird jede Permutation in eine RDQL-Query umgewandelt mit der jeder Query-Agent befragt wird. Zu unterscheiden sind hier zwei verschiedene Arten zur Erzeugung der RDQL-Queries. Einmal geschieht dies, indem eine RDQL-Query mit der Relation *"wirdUmschriebenDurch"* erzeugt wird. Zum anderen wird eine RDQL-Query mit der Relation *"wirdIdentifiziertDurch"* erzeugt wird. Die beiden unterschiedlichen RDQL-Queries werden von jeder Permutation sequentiell erzeugt. Wurde schließlich eine RDQL-Query formuliert, startet die Methode **rekSearch** ein Verhalten auf dem Agenten zum Versenden der RDQL-Query an die Query-Agents.

```
public void rekSearch(Vector searchStrings, boolean description) {
    Vector result = new Vector();
    String rdql = "";
    for (int i = 0; i < searchStrings.size(); i++) {
        String tmp = (String) searchStrings.get(i);
        Vector checkedWords = analyseSearchString(tmp);
        if (description) {
            rdql = generateRdqlString(checkedWords, this.DESCRIPTION);
        } else {
            rdql = generateRdqlString(checkedWords, this.IDENTIFY);
        }
        addBehaviour (new BehaviourQueryAnswerAgent (this, rdql));
    }
    Vector alternate = new Vector();
    for (int i = 0; i < searchStrings.size(); i++) {
        Vector newSelection =
            alternateSearchString(
                analyseSearchString((String) searchStrings.get(i)));
        if (newSelection == null) {
            return;
        }
        alternate.addAll(newSelection);
    }
```

```
    if (description) {
        rekSearch(alternate, this.DESCRIPTION);
    } else {
        rekSearch(alternate, this.IDENTIFY);
    }
    return;
}
```

In der Methode **generateRdqlString**, die von **rekSearch** aufgerufen wird, wird die eigentliche RDQL-Query formuliert, die zur Abfrage an die Query-Agents geschickt wird. Falls der Übergabeparameter *description* auf *true* gesetzt ist, wird eine RDQL-Query zur Suche in den beschreibenden Vokabularen generiert, ansonsten zur Suche in den identifizierenden Vokabularen. Mit *Config.BASE* wird auf den von der Ontologie verwendeten *Namespace* zugegriffen.

```
private String generateRdqlString(Vector searchWords,
                                  boolean description) {
    String requestString = "SELECT ?x WHERE ";
    String firstPart = new String();
    String secondPart = new String();
    int selectionIndex = 1;
    for (Iterator it = searchWords.iterator(); it.hasNext();) {
        firstPart += "(?sammlung" + selectionIndex + ",";
        secondPart += "(?x,";
        firstPart += "<" + Config.BASE + "#" + "umfasst>,";
        if (description) {
            secondPart += "<" + Config.BASE + "#"
                          + "wirdUmschriebenDurch>,";
        } else {
            secondPart += "<" + Config.BASE + "#"
                          + "wirdIdentifiziertDurch>,";
        }
        firstPart += "\"" + it.next() + "\")";
        secondPart += "?sammlung" + selectionIndex + ")";
        selectionIndex++;
    }
    requestString += (firstPart + secondPart);
    return requestString;
}
```

In den Methoden **alternateSearchString** und **analyseSearchString** wird der Suchstring permutiert und in einer neuen Reihenfolge zurückgegeben. Darauf gehe ich hier nicht näher ein, da es in der Arbeit von Herrn Pour-Heidari[24] ausführlich beschrieben wurde und ich die Methoden ohne Änderungen übernommen habe.

In der Methode **rekSearch** wird nach der Generierung einer neuen RDQL-Query ein Verhalten des Agenten gestartet, was eine Nachricht mit der Query

als Inhalt an die Antwort-Agenten verschickt. Das Verhalten *BehaviourQueryAnswerAgent* ist ein *OneShotBehaviour* und wird einmalig ausgeführt zur Übermittlung der Nachricht. In der Methode **action** des Verhaltens wird über die bibliothekseigene Klasse *Message* eine *QueryMessage* generiert mit der RDQL-Query als Inhalt. Es können hierbei beliebig viele Query-Agents die Query erhalten, indem einfach zusätzlich eine Nachricht an diese verschickt wird. Im folgenden Codebeispiel sieht man jedoch nur die Implementierung für einen Query-Agenten.

```
class BehaviourQueryAnswerAgent extends OneShotBehaviour {
    private String query = null;
    private Agent a;
    public BehaviourQueryAnswerAgent(Agent _a, String _query) {
        super(_a);
        a = _a;
        query = _query;
    }
    public void action() {
        send(Message.createQueryMessage(
                        a,
                       "QueryAgent",
                       query));
    }
}
```

6.6 Query-Agent

Im vorherigen Kapitel wurde beschrieben, wie der Broker-Agent vom Benutzer-Interface übergebene Suchstrings erhält, daraus RDQL-Queries formuliert und wie diese an die Query-Agents geschickt werden. In diesem Kapitel beschreibe ich die für den Prototypen relevanten Änderungen der Implementierung vom AnswerAgent aus der Demoversion von AgentOWL, ebenso wie relevante Funktionalitäten dieses Agenten innerhalb des Agenten-Systems.

Beim Starten des Query-Agents wird, genau wie beim Broker-Agent, die Konfiguration, sowie die Ontologie in den Speicher des Agenten geladen. Die Ontologie des Query-Agents hat jedoch sehr wohl Instanzen im Gegensatz zum Broker-Agent. In der Ontologie des Query-Agents sind die eigentlichen Informationen gespeichert nach denen gesucht werden kann.

Nachrichten empfängt der Query-Agent über das Verhalten *BehaviourHandleRecivedMessages*. Der Inhalt der Nachricht, die RDQL-Query, wird extrahiert, um dann die Abfrage gegen die Ontologie über den Jena-eigenen Reasoner zu starten. Jede Ressource, die als Ergebnis dieser Abfrage vom Reasoner zurückgegeben wird, schickt der Query-Agent über die bibliothekseigene Klassenmethode *Message.createInformMessage* an den Broker-Agent

zurück.

```
class BehaviourHandleReceivedMessages extends CyclicBehaviour {
    public BehaviourHandleRecivedMessages(Agent _a) {
        super(_a);
    }
    public void action() {
        synchronized(this) {
        ACLMessage msg = receive();
        if (msg != null) {
            case ACLMessage.QUERY_REF:
            if (msg.getLanguage().equals(Ontology.RDQL)) {
                Model m = mem.getModel();
                Query q = new Query(msg.getContent()) ;
                q.setSource(m);
                QueryExecution qe = new QueryEngine(q);
                QueryResults res = qe.exec() ;
                for ( Iterator iter = res ; iter.hasNext() ; ) {
                   ResultBinding resB = (ResultBinding)iter.next();
                   Resource x = (Resource)resB.get("x") ;
                   send(Message.createInformMessage(
                               this.myAgent,
                               "AskAgent",
                               x));
                }
                res.close() ;
            }
        }
        block();
    }
}
```

Die Abfrage gegenüber dem Jena-eigenen Reasoner erfolgt zunächst über das Initialisieren des Reasoners mit dem Ontologiemodell, was im Speicher des Agenten hinterlegt ist mit

```
Model m = mem.getModel();
```

Nun wird die Query auf das Modell gelegt mit

```
Query q = new Query(msg.getContent())} und
q.setSource(m);
```

Damit kann der Reasoner gestartet werden mit

```
QueryExecution qe = new QueryEngine(q);
QueryResults res = qe.exec();
```

6.7 Die Topologie des Multi-Agentensystems

Die im vorangegangenen beschriebenen Agenten bilden gemeinsam die Komponenten der Topologie des Multi-Agentensystems. Es existieren Broker-Agenten, die jeweils über ein Benutzer-Interface angesprochen werden und Query-Agenten, an die die Suchanfragen delegiert werden. Exemplarisch möchte ich in diesem Kapitel die Topologie mit einem Benutzer-Interface, einem Broker-Agent und zwei Query-Agents beschreiben. Diese Topologie als Beispiel ist in Abbildung 19 zu sehen. Die zwei Query-Agents arbeiten auf zwei unterschiedlichen Versionen der Haupt-Ontologie:

- Eine Version, die nur Instanzen des Typs *Dienstleister* beinhaltet
- und eine Version, die nur Instanzen des Typs *Dienstleistung* beinhaltet.

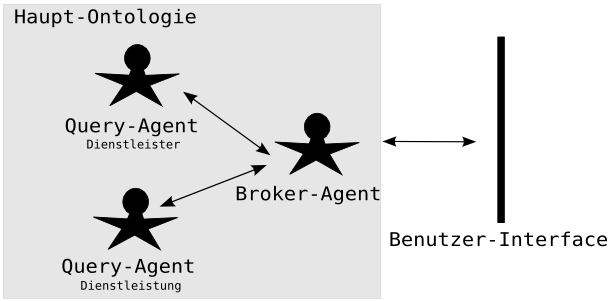

Abbildung 19: Topologie

Jede der beiden verschiedenen Ontologie-Versionen läuft auf einem eigenen Query-Agent. Hierfür wird im Query-Agent der Name des Agenten angegeben und die zu ladende Konfiguration inklusive Ontologieversion. Der Agent *AnswerDienstleistung*, der in seiner Ontologieversion nur Instanzen vom Typ *Dienstleistung* enthält, läd dafür seine eigene OWL-Datei beim Start, ebenso wie der Agent *AnswerDienstleister* seine OWL-Datei mit der Ontologieversion, die nur Instanzen des Typs *Dienstleister* enthält. Der Vorteil bei der Verteilung der Ontologie-Versionen auf verschiedene Agenten liegt darin, dass direkt auf den Peers die Suche in den Ontologien stattfinden kann.

Damit die Anfragen des Broker-Agents an beide Query-Agents geschickt werden können, sind lediglich in dem Verhalten *BehaviourQueryAnswerAgent* die beiden Query-Agents mit ihrem neuen Namen einzutragen:

```
send(Message.createQueryMessage(
                a,
                "AnswerDienstleister",
```

```
                query));
send(Message.createQueryMessage(
                a,
                "AnswerDienstleistung",
                query));
```

Nachdem man die Ontologien in verschiedene Versionen mit unterschiedlichen Instanzen aufgesplittet hat, ist es möglich diese noch weiter zu differenzieren und weitere mit ihnen verbundene Query-Agents in das System zu integrieren. Das einzige, was zu beachten ist, ist, dass eine Haupt-Ontologie innerhalb des Agentensystems verwendet wird. Auf welchem Query-Agent die Instanzen innerhalb der Ontologie nachher verwaltet werden, ist dann transparent für den Benutzer. Über das Benutzer-Interface wird eine Suchanfrage abgegeben und der Broker-Agent delegiert die aufbereiteten RDQL-Queries an die Query-Agenten weiter, die bei ihm nur in die Empfängerliste eingetragen werden müssen.

7 Fazit

In dieser Diplomarbeit wurde die Entwicklung eines Prototypens einer ontologiebasierten Suche in einem MAS beschrieben. Die dafür verwendeten Technologien sind die Frameworks JADE und Jena.

Losgelöst von Datenstrukturen unter Verwendung von Ontologien zur Beschreibung von semantischen Zusammenhängen ist es möglich, sehr komplexe thematische Anwendungsgebiete abzubilden. Es ist nicht mehr notwendig, sich bei einer solchen Modellierung um die konkrete architekturabhängige Speicherung der Informationen zu kümmern. Diese sonst so investierte Zeit zur Implementierung komplexer Datenstrukturen kann dafür genutzt werden, an den Problemfall angelehnte Strategien zur Abbildung von Informationszusammenhängen in Modellen einzusetzen.

Durch die Verwendung von Agenten als Komponenten innerhalb des Architekturmodells ist es möglich, die einzelnen Knoten innerhalb des Systems als in sich geschlossene Einheiten zu betrachten, und sie so auch zu implementieren. Ein Knoten bildet eine autonome Komponente. Es ist nicht mehr notwendig, das System als gesamtes bei der Implementierung zu betrachten. So ist das System sehr leicht skalierbar, und zusätzliche Knoten können problemlos entfernt oder hinzugefügt werden. Dadurch dass die Haupt-Ontologie in verschiedenen Versionen auf die Query-Agenten verteilt werden kann, wird direkt auf den Peers gesucht. Man hat keine zentrale Suchmaschine mehr, die sämtliche Anfragen entgegen nimmt und in ihrer Datenbank sucht, sondern die Suche wird über die Broker-Agenten an die Query-Agenten delegiert. Der Benutzer kontaktiert einen Broker-Agenten, der autonom die in Frage kommenden Query-Agenten kontaktiert und die Ergebnisse aufbereitet an den Benutzer zurückgibt. Die Query-Agenten können sich frei im Netz bewegen während sie Queries erhalten, um Anfragen auf ihren Datenbestand abzusetzen.

Die exemplarische Implementierung des Prototypen einer ontologiebasierten Suche in einem Multi-Agentensystem verdeutlicht die Möglichkeit der Kombination der Technologien *Ontologien* und *Agenten*.

Problematisch ist die Entwicklung der Ontologie für ein solches Suchsystem, da die gesamten Suchfunktionen auf die Ontologie aufbauen. Die Implementierung der Algorithmen zur Erstellung der RDQL-Queries in dem Prototypen ist eine proprietäre Lösung, da diese an die verwendete Ontologie angepasst sind und darauf aufbauen, dass *umschreibendeBegriffe* und *identifizierende Begriffe* in der Ontologie zur Identifizerung der jeweiligen Instanz vorhanden sind. Die geladene Ontologieversion des Agentenspeichers ist zwar austauschbar, aber es ist nicht möglich die Ontologie auszutauschen, und auf

dieser dann die Suche laufen zu lassen.

In Kapitel 8 beschreibe ich Ausblicke auf die Möglichkeiten zukünftiger Entwicklungen.

8 Ausblick

Zur Implementierung des Prototypen einer ontologiebasierten Suche in einem Multi-Agentensystem, wurde nur eine Ontologie entwickelt, auf die der Prototyp angepasst ist. Es ist also eine proprietäre Lösung, in der das Agentensystem an die verwendete Ontologie gekoppelt ist.

Eine Möglichkeit für weitere Entwicklungen wäre es, weitere ähnliche Ontologien zu entwickeln auf denen gesucht werden kann, und den Prototypen dahin gehend zu erweitern, dass er auf diesen anderen Ontologien suchen kann. Anschließend könnte man versuchen die unterschiedlichen ontologiebasierten Agentensysteme zu koppeln, so dass über eine Benutzeranfrage in den verwandten und ähnlichen Ontologien gesucht werden kann.

Solche eng verwandten Ontologien werden unter einer Domäne zusammen gefasst. Hat man die Implementierung zu Suchfunktionen in verschiedenen Ontologien innerhalb einer Domäne implementiert, könnte man dazu übergehen, verschiedene Domänen zu entwickeln, und diese miteinander zu verknüpfen. Mit der Verknüpfungen von mehreren Ontologien innerhalb verschiedener Domänen beschäftigt sich seit 1996 eine Forschungsgruppe am Massachussettes Institute of Technology unter dem Projektnamen **D**istributed **O**ntology **M**anagement **E**nvironmet, kurz DOME [25]. Zur Implementierung von Suchfunktionen innerhalb verschiedener Domänen- Ontologien, könnte der in dieser Diplomarbeit entwickelte Prototyp einen ersten Ansatz bieten.

Der im Rahmen dieser Diplomarbeit entwickelte Prototyp wurde als experimenteller Prototyp entwickelt, um zu erproben, ob die Kombination der beiden Technologien Ontolgien und Agenten realistisch ist und einen Ansatz bietet für Entwicklungen in dieser Richtung. Jedoch sollte die Praxisnähe bei der Entwicklung weiterer Prototypen beachtet werden. Das hier entwickelte System sollte aus dem Kontext eines experimentellen Prototypen in die Richtung eines evolutionären Prototypen gehen, der eng an Anwendungsfälle orientierte Problemlösungen darstellt. Ontologien spiegeln ein punktuelles Abbild von Zusammenhängen der Realität dar, die schlichtweg unbrauchbar sein können, falls die Abbildung der Realität in der Ontologie nicht stimmig ist. Die Gefahr bestünde sonst ein System zu entwickeln, was eine fiktive Realität modelliert und Problemlösungen implementiert, die einfach nicht stimmen, wenn es zu konkreten Anwendungsfällen in einem zu späten Entwicklungszyklus kommt.

Literatur

[1] Schahram Dustdar, Harald Gall, Manfred Hauswirth. Software- Architekturen für Verteilte Systeme. *Beispiele für Peer- to Peer- Architekturen.* Springer Verlag, Juli 2003.

[2] Tim Berners- Lee, James Hendler, Ora Lassila. Spektrum der Wissenschaft. *Mein Computer versteht mich.* Akademischer Verlag, August 2001. S. 42.

[3] Michael Kuschke, Ludger Wölfel. Web Services Kompakt. Spektrum Akademischer Verlag, Heidelberg Berlin 2002.

[4] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard. Web Services Architecture. *What is a Web service?* http://www.w3c.org/TR/ws-arch/#id2260073. W3C Working Group Note, 11. Februar 2004.

[5] Stan Franklin, Art Graesser. Is it an agent, or just a program? *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages.* Springer- Verlag, 1996, S. 1-4.

[6] Sankar Virdhagriswaran, Damian Osisek, Pat O'Connor. Standardizing agent technology. *StandardView,* ACM Press, 1995, S. 96-101.

[7] Stuart Russel, Peter Norvig. Intelligent Agents. *Artificial Intelligence: A Modern Approach.* 1995, S. 33.

[8] Pattie Maes. Artificial Life Meets Entertainment: Life Like Autonomous Agents. *Communications Of The ACM.* ACM Press, Volume 38, 1995, S. 108.

[9] D.C. Smith, A. Cypher, J. Spohrer. KidSim: Programming Agents Without A Programming Language. *Communications Of The ACM.* ACM Press, Volume 37, 1994, S. 55-67.

[10] B. Hayes- Roth. An Architecture for Adaptive Intelligent Systems. *Artificial Intelligence: Special Issue on Agents and Interactivity.* 1995, S. 329-265.

[11] Michael Wooldridge, Nicholas R. Jennings. Agent Theories, Architectures, and Languages: a Survey. *Intelligent Agents.* Springer- Verlag, 1995, S. 2.

[12] Jose C. Brustolini. Autonomous Agents: Characterization and Requirements. *Carnegie Mellon Technical Report CMU-CS-91-204.* Carnegie Mellon University, Pittsburgh, 1991, S. 265.

[13] Nicholas R. Jenninggs, Katia Sycara, Michael Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems. 7-37.* Kluwer Academic Publisher, Boston 1998. S. 2-3.

[14] *Multiagentensystem* http://de.wikipedia.org/wiki/Multiagentensystem

[15] Jürgen Quade. Linux- Treiber entwickeln, Gerätetreiber für Kernel 2.6 systematisch eingeführt. *Hotplug.* http://ezs.kr.hsnr.de/TreiberBuch/html/index.html. November 2005.

[16] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB, vorher CSELT), Giovanni Rimassi (Universität Parma, Jade Programmer's Guide. *The agent tasks. Implementing Agent behaviours.* TILab S.p.A., 2004. S. 23-24.

[17] http://www.fipa.org *Welcome to FIPA!* Foundation for Intelligent Physical Agents, 2005.

[18] http://de.wikipedia.org/wiki/Ontologie_%28Informatik%29 *Ontologie (Informatik).* Wikipedia, 2005.

[19] Bernd Oestereich. Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language. *Einführung.* Oldenbourg Verlag, München, Wien, 2001. S. 28.

[20] Jean Vaucher and Ambroise Ncho. JADE Tutorial and Primer. *Using ontologies.* Fakultät für Informatik, Universität Montreal, September 2003, aktualisiert April 2004

[21] The Protege Ontology Editor and Knowlege Acquisition System. http://protege.standford.edu.

[22] Erich Gamma, Richard Helm, Ralph E. Johnson. Entwurfsmuster. *Verhaltensmuster.* Addison- Wesley, München, Juli 2004. S. 287-301.

[23] Michal Laclavik. AgentOWL. *Agent Library for Supporting RDF/ OWL model based on Jena.* http://ups.savba.sk/%7Emisos/AgentOWL/doc/index.html. Institute of Informatics SAS, Bratislava. Slowakei. Januar 2005.

[24] Amir Pour- Heidari. *Ontologiegestützte Websuche.* Fachhochschule Aachen, 2005.

[25] Distributed Ontology Management Environment. *http://dome.sourceforge.net* Massachussettes Institute of Technology, seit 1996.