Seminararbeit

Thema

Softwaretechnologie für Agenten

im Rahmen des Informatik-Seminars „Methoden der KI“ bei Prof. Dr. Sebastian Iwanowski im Sommersemester 2010

Erarbeitet von

Jan Ahrens

Inhaltsverzeichnis

Abbildungsverzeichnis III

Abkürzungsverzeichnis IV

1 Einführung 1
- 1.1 Agenten 1
- 1.2 Multiagenten-Systeme 1
- 1.3 Kommunikation 2
  - 1.3.1 Sprechakttheorie 2
  - 1.3.2 Interaktionsprotokolle 3
- 1.4 Repräsentation von Wissen 4

2 Softwaretechnologie 6
- 2.1 Architektur 6
- 2.2 Das JADE Framework 6
- 2.3 Agenten in JADE 9
- 2.4 Beispiel einer Verhandlung mit JADE 11
  - 2.4.1 Der Händler Agent 11
  - 2.4.2 Der Kunden Agent 12
  - 2.4.3 Besonderheiten bei der Verarbeitung von Nachrichten 14
- 2.5 Ontologien und Content-Sprachen in JADE 14
- 2.6 Weitere Technologien 17

3 Einschätzung und Bewertung 18
- 3.1 Schwächen von JADE 18
- 3.2 Stärken von JADE 18
- 3.3 Multiagenten-Systeme 19

Literaturverzeichnis 20

Abbildungsverzeichnis

1.1 Eine Contract Net Verhandlung 4
1.2 Die Hotel-Gast-Ontologie 5

2.1 FIPA Agenten-Plattform 7
2.2 Eine komplexere JADE Konfiguration 8
2.3 Der Haupt-Container von „mars“ 9
2.4 Ausschnitt aus dem UML-Diagramm für JADE Agenten 10
2.5 Ablauf eines Agenten 10
2.6 Die Content-Sprachen Ontologie Pipeline 16

Abkürzungsverzeichnis

AAMAS	Autonomous Agents and Multiagent Systems.
AID	Agent Identifier.
AMS	Agent Management System.
AP	Agent Platform.
API	Application Programming Interface.
BDI	Belief Desire Intention.
DF	Directory Facilitator.
FIPA	Foundation for Intelligent Physical Agents.
HTTP	Hypertext Transfer Protocol.
IRE	Identifying Referential Expression.
JADE	Java Agent DEvelopment.
JVM	Java Virtual Machine.
MTP	Message Transport Protocol.
MTS	Message Transport Service.
RMA	Remote Management Agent.
SL	Semantic Language.

1 Einführung

1.1 Agenten

Agenten sind eine spezielle Klasse von Computerprogrammen, die wie folgt definiert werden [1]:

> Ein Agent ist ein Computerprogramm das sich in einer bestimmten Umgebung befindet und welches fähig ist, eigenständige Aktionen in dieser Umgebung durchzuführen, um seine Ziele zu erreichen.

Die Aktionen eines Agenten werden meistens ausgeführt, um Ziele eines Auftraggebers zu erreichen. Ein solcher Agent kann als Assistent betrachtet werden.

In der Literatur existieren weitere Definitionen des Agenten-Begriffs, die einige zusätzliche Eigenschaften fordern. Die hier genannte Definition ist allgemein und trifft auf eine große Klasse von Agenten zu. Für die Konstruktion von Agenten werden folgende Eigenschaften gefordert.

- Autonomie
- Proaktivität
- Reaktivität

Ein autonomer Agent hat Kontrolle über seinen eigenen Zustand und sein Verhalten. Proaktive Agenten sind in der Lage ihre Ziele durch Eigeninitiative zu erreichen. Durch die Eigenschaft der Reaktivität, wird es Agenten ermöglicht auf Reize aus der Umwelt zu reagieren. Diese Reize können unter anderem Aufträge eines Auftraggebers sein oder Veränderungen des Zustands der Umwelt.

Um den Begriff des Agenten etwas plastischer zu gestalten, soll folgender Anwendungsfall aus dem Umfeld des semantischen Web dienen:

> Um einen Urlaub zu buchen, wird einem persönlichen Web-Agenten der Auftrag gegeben eine Unterkunft auf Mallorca zu buchen. Der Agent tritt in Verhandlung mit den Agenten von unterschiedlichen Hotel-Anbietern und handelt nach den Vorlieben seines Auftraggebers das beste Angebot aus, um daraufhin das Hotel zu buchen.

Es ist erkennbar das ein Agent in diesem Fall nicht allein handelt, sondern erst durch die Zusammenarbeit mit anderen Agenten seine Ziele erreichen kann. Dies führt zum Begriff der Multiagenten-Systeme.

1.2 Multiagenten-Systeme

Oft ist der Einsatz eines Agenten nicht ausreichend um komplexe Problemstellungen zu lösen. Werden mehrere unabhängige Agenten vernetzt, so spricht man von einem Multiagenten-System. Zur Problemlösung besitzen Agenten zusätzliche soziale Eigenschaften, die eine Interaktion untereinander ermöglichen.

Um den Begriff des Multiagenten-Systems, vom Begriff des verteilten Systems abzugrenzen, ist es wichtig zu wissen, dass die Agenten unterschiedliche Ziele verfolgen und zu unterschiedlichen Organisationen gehören können.

Multiagenten-Systeme liefern eine Reihe von zusätzlichen Problemstellungen, die gelöst werden müssen:

1. Agenten müssen kommunizieren, um zu kooperieren.

2. Agenten müssen sich über ihr Wissen austauschen.

3. Agenten müssen sich untereinander kennen, um zu kommunizieren.

Das Problem der Kommunikation wird im Abschnitt 1.3 erklärt. Die Repräsentation des Wissens wird im gleichnamigen Abschnitt 1.4 behandelt. Mit dem Problem der Agenten Vermittlung, beschäftigt sich der Abschnitt 2.1.

1.3 Kommunikation

Damit ein Agent in einem Multiagenten-System seine Ziele erreichen kann, besteht die Notwendigkeit zum Austausch von Wissen und zur Delegation von Aufgaben. Beides kann nur durch Kommunikation der Agenten untereinander erreicht werden.

Im Gegensatz zur uns bekannten sprachlichen Kommunikation, erfolgt die Kommunikation von Agenten stets asynchron. Der Grund hierfür ist technisch bedingt: Jede Antwort braucht eine gewisse Zeit und ein Agent sollte gleichzeitig mehrere Konversationen halten können. Würde die Agenten Kommunikation unter diesen Bedingungen synchron ablaufen, so würde der Agent blockiert werden und könnte nicht so effizient arbeiten, wie mit asynchroner Kommunikation.

Da in einem Multiagenten-System Agenten unterschiedlicher Anbieter miteinander kommunizieren müssen, ist eine Standardisierung der Kommunikation erforderlich.

Um die Interoperabilität von Agenten-Systemen im kommerziellen und industriellen Umfeld zu fördern, wurde 1996 die Foundation for Intelligent Physical Agents (FIPA) gegründet. Die FIPA ist eine zur „IEEE Computer Society“ gehördende Standardisierungsorganisation mit Sitz in der Schweiz.

Mit der „FIPA Communicative Act Library Specification“ [2], wurde von der FIPA einen Standard herausgegeben, der die Kommunikation zwischen Agenten auf Grundlage von Sprechakten modelliert.

1.3.1 Sprechakttheorie

Die Sprechakttheorie wurde 1962 von John Langshaw Austin begründet. In „How to Do Things with Words“ [3] beschäftigt sich Austin mit der Frage wie Sprache verwendet wird um alltägliche Absichten und Ziele zu erreichen. Er beobachtet, dass durch Äußerungen nicht nur Sachverhalte beschrieben und Behauptungen aufgestellt, sondern auch Handlungen (Sprechakte) vollzogen werden (vgl. [4]). Diese Klasse von Äußerungen werden performative Äußerungen (engl. performatives) genannt. Als Beispiele nennt Austin eine Kriegserklärung oder den Satz „Hiermit erkläre ich euch zu Mann und Frau“. Durch performative Äußerungen wird der Zustand der Umwelt verändert.

Auch durch die Agenten-Kommunikation sollte die Umwelt verändern werden, da das Erreichen eines Ziels eine Veränderung des Zustands der Agenten-Umwelt bedeutet. Aus diesem Grund wird die Kommunikation zwischen Agenten durch performative Äußerungen aus der Sprechakttheorie modelliert.

In der „Communicative Act Library Specification“ [2] spezifizert die FIPA 22 performative Verben. Einige Beispiele dieser Verben sind:

- „query“, zur Formulierung von Fragen
- „inform“, zum Mitteilen von Fakten
- „request“, zur Verteilung von Aufgaben
- „not-understood“, um mitzuteilen das eine Nachricht nicht verstanden wurde

Alle Verben lassen sich in der Spezifikation einsehen, wo diese formal definiert werden.

1.3.2 Interaktionsprotokolle

Aufbauend auf der „Communicative Act Library Specification“ [2], wurden von der FIPA neun Interaktionsprotokolle spezifiziert, mit deren Hilfe eine geordnete Agenten-Kommunikation erfolgen kann. Die Protokolle sind in folgenden FIPA Standards spezifiziert.

- Request Interaction Protocol (SC00026)
- Query Interaction Protocol (SC00027)
- Request When Interaction Protocol (SC00028)
- Brokering Interaction Protocol (SC00033)
- Recruiting Interaction Protocol (SC00034)
- Subscribe Interaction Protocol (SC00035)
- Propose Interaction Protocol (SC00036)
- Contract Net Interaction Protocol (SC00029)
- Iterated Contract Net Interaction Protocol (SC00030)

Das „Request Interaction Protocol“ spezifiziert beispielsweise wie die Delgation einer Aufgabe mit Rückmeldungen abläuft. Alle Interaktionsprotokolle sind auf der Internetseite der FIPA[1] einsehbar.

Das Contract Net Protokoll [5] wird im Rahmen dieses Vortrags genauer erläutert, da im Verlauf dieses Arbeit ein Software-Beispiel mit Contract Net realisiert werden soll (siehe Abschnitt 2.4).

Ziel des Protokolls ist die Verteilung einer Ressource (oder Aufgabe). Der Ablauf des Protokolls ist in Abbildung 1.1 grafisch dargestellt. Um die Lesbarkeit zu erhöhen, wurden einige Nachrichten weggelassen.

Der Ablauf beginnt damit das ein Manager die Ressource im Rahmen einer Auktion ausschreibt und Nachrichten an alle Teilnehmer sendet (call-for-propose). Die Teilnehmer können sich darauf hin entscheiden ob sie ein Angebot abgeben möchten (propose) oder die Abgabe eines Angebots ablehnen (reject-propose [nicht in der Grafik dargestellt]). Versteht ein Agent nicht was mit der Ausschreibungsnachricht (call-for-propose) gemeint ist, so reagiert er mit einer Nicht-Verstanden-Nachricht[2] (not-understood [nicht in der Grafik dargestellt]). Der Manager der Auktion wartet auf eine Antwort von allen Teilnehmern. Wenn dem Manager alle Angebote vorliegen, wählt dieser das nach seinen Kriterien beste Angebot aus und sendet diesem Agenten eine Annahme-Nachricht (accept-proposal).

[1] `http://www.fipa.org/repository/standardspecs.html`

[2] Dieses Verhalten wird von Agenten allgemein genutzt, wenn sie eine Nachricht nicht verstanden haben.

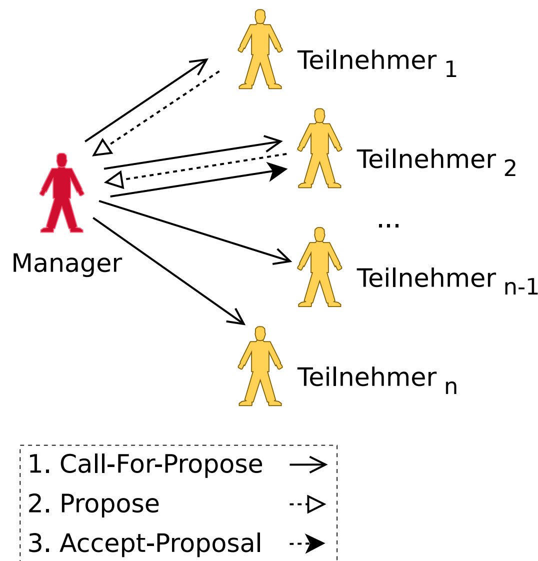

Abbildung 1.1: Eine Contract Net Verhandlung

Damit die Auktion abgeschlossen werden kann, muss der Agent, der vom Manager den Zuschlag erhalten hat, eine Bestätigung an den Manager schicken (inform) oder ihn darüber informieren das die Zuweisung des Auftrag fehlgeschlagen ist (failure). Der letzte Schritt ist notwendig, da ein Agent gleichzeitig an beliebig vielen Auktionen teilnehmen kann und die Möglichkeit besteht das er bereits in einer anderen Auktion einen Zuschlag erhalten hat.

In der iterativen Variante des Contract Net Protokolls (Iterated Contract Net), hat der Manager nach Erhalt aller Angebote die Möglichkeit eine neue Ausschreibung an die Teilnehmer zu senden, basierend auf den bereits erhaltenen Angeboten. Diese Abwandlung erlaubt es komplexere Verhandlungssituationen abzubilden, ist jedoch nicht in allen Fällen sinnvoll. Man kann sich vorstellen, dass ein Manager eine Aufgabe versteigert und die Teilnehmer mit der verfügbaren Rechenzeit bieten. Eine Iteration ist in diesem Fall nicht sinnvoll, da im allgemeinen Fall den Agenten nicht mehr Rechenzeit bei einer zweiten Ausschreibungsrunde zur Verfügung steht.

1.4 Repräsentation von Wissen

Im Abschnitt 1.3 wurde die Kommunikation zwischen Agenten behandelt. Damit eine Kommunikation zwischen Agenten sinnvoll ablaufen kann, müssen diese sich verstehen. Dazu müssen die Inhalten der Kommunikationsnachrichten eine gewisse Bedeutung haben.

Ontologien helfen die Semantik von Nachrichten zu definieren. Sie werden in der Wikipedia wie folgt definiert[6]:

> Ontologien [...] sind [...] formal geordnete Darstellungen einer Menge von Begrifflichkeiten und der zwischen ihnen bestehenden Beziehung in einem bestimmten Gegenstandsbereich.

Außer der Repräsentation von Wissen, bieten Ontologien die Möglichkeit zum Erschließen von neuem Wissen. Dies geschieht mit Inferenz-Verfahren (logisches Folgern). Im Rahmen dieser Arbeit werden Ontologien jedoch ausschließlich zur Wissensrepräsentation verwendet und daher wird im Folgenden nicht weiter auf das Erschließen von Wissen eingegangen.

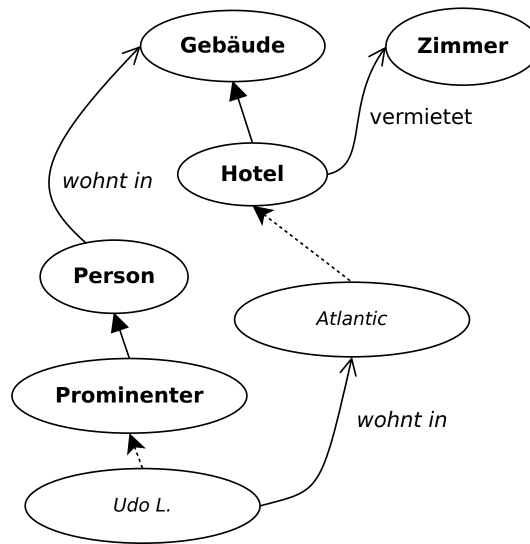

Abbildung 1.2: Die Hotel-Gast-Ontologie

Um den Begriff der Ontologie zu verdeutlichen und ein Anwendungsbeispiel zu geben, wird folgendes Beispiel verwendet. In einem Hotel-Umfeld werden Nachrichten über Gäste und Buchungen ausgetauscht. Damit alle Agenten die Bedeutung der Nachrichten verstehen können, wirde folgende Ontologie definiert (siehe Abbildung 1.2):

- Ein **Hotel** ist ein **Gebäude**.
- Ein **Prominenter** ist eine **Person**.
- Ein Hotel **vermietet** mehrere **Zimmern**.
- Eine **Person** kann in einem **Gebäude wohnen**.
- Das **Atlantic** ist ein **Hotel**.
- **Udo L.** ist ein **Prominenter**.
- **Udo L.** wohnt im **Atlantic**.

Die Ontologie besteht aus Konzepten (Hotel, Gebäude, Prominenter, Person), Vererbungen (Hotel ist Gebäude, Promienter ist Person), Prädikaten (vermietet, wohnt in) und Instanzen („Udo L.“ und ‘Ätlantic“).

Mit Hilfe dieser Ontologie kann ein Agent die Frage stellen: Ist es wahr, dass „Udo L.“ im „Atlantik“ wohnt (bezogen auf Prädikat „wohnt in’”). Der Empfänger dieser Frage könnte ein Hotelverwaltungsagent des „Atlantik“ sein, der diese Frage versteht, da sie laut der Ontologie semantisch korrekt ist, und eine Antwort aufgrund seiner Wissensdatenbank geben kann.

Würde jedoch als Inhalt einer Nachricht die Aussage formuliert, dass das „Atlantik“ „Udo L.“ vermietet, könnte jeder Agent, der diese Ontologie kennt, feststellen, dass diese Nachricht semantisch nicht korrekt ist. Das „Atlantik“ ist zwar ein „Hotel“ und kann daher „Zimmer“ vermieten, doch „Udo L.“ ist eine Instanz des Konzeptes „Prominenter“, welches nicht von „Zimmer“ erbt (vgl. Abbildung 1.2).

2 Softwaretechnologie

2.1 Architektur

In der „FIPA Agent Management Specification“ [7] wird von der FIPA ein Referenz-Modell für eine Agenten-Plattform (Agent Platform (AP)) spezifiziert. Eine Agenten Plattform stellt eine Middleware für die Ausführung von Agenten bereit. Sie ist modular aufgebaut und besteht aus drei Hauptkomponenten (siehe Abbildung 2.1).

Das Message Transport Service (MTS) sorgt dafür, dass Nachrichten zwischen Agenten ausgetauscht werden können. Nachrichten können mit unterschiedlichen Message Transport Protocol (MTP) versendet werden, die vom MTS bereitgestellt werden. Ein Beispiel für ein MTP ist das aus dem Web-Umfeld stammende Hypertext Transfer Protocol (HTTP).

Das Agent Management System (AMS) ist für den Betrieb der Agenten Plattform zuständig. Es überwacht den Lebenszyklus der Agenten und verwaltet ihren Status. Zusätzlich stellt das AMS sicher, dass die Agenten mit einem eindeutigen Agent Identifier (AID) angesprochen werden können.

Eine optionale Komponente der Agenten Plattform ist der Directory Facilitator (DF), welcher eine Art Branchenbuch zur Verfügung stellt. Jeder Agent, der einen bestimmten Dienst zur Verfügung stellt, kann sich beim DF registrieren. Sucht ein Agent andere Agenten, die ein speziellen Dienst bereitstellen, so kann dieser eine Anfrage an den DF richten. Der DF löst das Problem der Agenten Vermittlung aus Abschnitt 1.2.

Über das MTS können Agenten einer Plattform mit Agenten anderer Plattformen kommunizieren. Da neben dem AMS auch der DF der Plattform als Agent realisiert ist, besteht die Möglichkeit auch Agenten anderer Plattformen mit ihren Diensten zu finden, indem der DF Agent einer anderen Plattformen angefragt wird.

2.2 Das JADE Framework

Das Java Agent DEvelopment (JADE) Framework, stellt eine Middleware und Bibliotheken für die Entwicklung von Multiagenten-Systeme bereit. Es wurde ursprünglich von der Telecom Italia entwickelt und wird unter einer Open Source Lizenz[1] zur Verfügung gestellt. Bei der Entwicklung wurde Wert auf Kompatibilität zu den FIPA Spezifikationen gelegt, so dass die „JADE Plattform“ Agenten weiterer Plattformen ausführen kann und JADE Agenten auf anderen FIPA kompatiblen Plattformen lauffähig sind.

Um die JADE Plattform zu starten, genügt es die Klasse „jade.Boot“ aus dem „jade.jar“ Bibliothek auszuführen [2] (siehe Listing 2.1). Um einen eindeutigen Agenten Bezeichner bilden zu können, braucht jede Agenten Plattform einen eindeutigen Namen, der zusammen mit dem Agenten Namen nach dem folgen Muster konkateniert wird: <Agenten Name>@<Plattform Name>.

Beim Start einer Plattform muss ihr Name mit dem Parameter „-platform-id“ angegeben werden.

---

[1] JADE steht unter der Lesser GNU General Public License Version 2

[2] Alle Erklärungen und Quelltextbeispiele beziehen sich auf die JADE Version 4.0 vom April 2010, die unter von der JADE Seite unter `http://jade.tilab.com` bezogen werden kann.

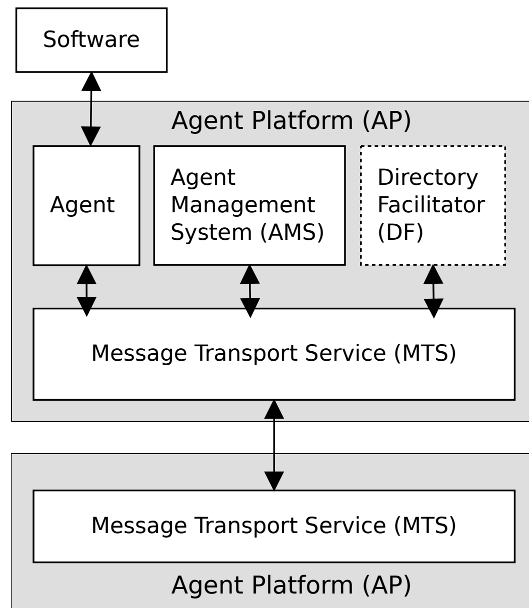

Abbildung 2.1: FIPA Agenten-Plattform

```
1  java -cp lib/jade.jar jade.Boot -platform-id mars
```

Listing 2.1: Start einer JADE Agenten Plattform mit Namen „mars“

Mit dem Remote Management Agent (RMA) stellt JADE einen speziellen Agenten bereit, welcher eine grafische Verwaltungsoberfläche für die Agenten Plattform bietet. Er lässt sich starten, indem der „-gui“ Parameter beim Start der Plattform genutzt wird. Alternativ lässt sich der RMA wie jeder andere Agent explizit startet (siehe Listing 2.2 und 2.3).

```
1  java -cp lib/jade.jar jade.Boot -platform-id mars -gui
```

Listing 2.2: JADE Plattform mit RMA starten

```
1  java -cp lib/jade.jar jade.Boot -platform-id mars rma:jade.tools.rma.rma
```

Listing 2.3: JADE Plattform mit RMA starten (explizite Angabe)

Nach Start des RMA wird sichtbar das eine JADE Plattform in Container unterteilt ist. Dieses Konzept hilft bei der Strukturierung der Agenten einer Plattform. Ein Container bildet immer eine Java Virtual Machine (JVM) ab. Eine JVM kann ebenfalls auf anderen Servern laufen. Es ist somit möglich Agenten unterschiedlicher Server zu einer Plattform zu verbinden.

Ein spezieller Container, der Haupt-Container (Main-Container), beinhaltet immer die Plattform Agenten AMS und DF. Werden Agenten zusammen mit der Plattform gestartet, so sind diese auch Teil des Haupt-Containers. Wird der RMA zusammen mit der Plattform gestartet (z.B. über den „-gui“ Parameter), ist diese Gegebenheit erkennbar.

Das Konzept der Container ist nicht dem FIPA Standard entnommen, sondern wurde von JADE hinzugefügt. Durch die FIPA Kompabilität von JADE werden alle Agenten einer Plattform jedoch nach außen ohne Container repräsentiert.

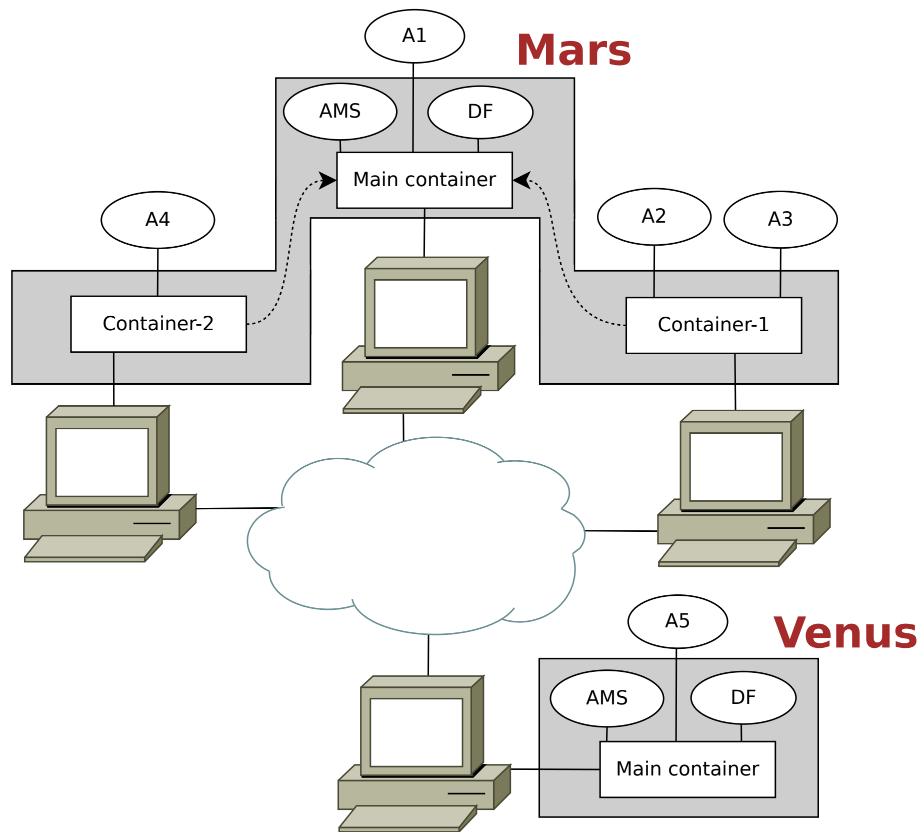

Abbildung 2.2: Eine komplexere JADE Konfiguration

Mit Hilfe des Container-Konzeptes lassen sich komplexe Plattform-Szenarien realisieren. Um dies zu verdeutlichen, wird eine Beispiel Konfiguration herangezogen, die auch in den offiziellen JADE Dokumentationen verwendet wird [8].

In Abbildung 2.2 werden zwei Plattformen dargestellt: „mars“ und „venus“. Jede Plattform enthält einen Main-Container, welcher die JVM darstellt, auf der die jeweilige Plattform gestartet wurde. Zusammen mit dem Main-Container von „mars“ wurde der Agent „A1“ gestartet und im Main-Container der Plattform „venus“ befindet sich der Agent „A5“. Die Plattform „mars“ besteht aus zwei weiteren Containern, die in diesem Beispiel auf zwei weiteren Servern laufen und ebenfalls Agenten enthalten.

Wie zu erkennen ist, lassen sich in einem Container beliebig viele Agenten starten, die sich alle am AMS des Main-Containers anmelden.

Um eine Zusammenarbeit zwischen Plattformen zu ermöglichen, können die Plattformen verknüpft werden. Hierzu kann das AMS einer Plattform mit dem AMS einer anderen Plattform verbunden werden. Die DF der einzelnen Plattformen, lassen sich ebenfalls verknüpfen (DF federation). Näheres findet sich im „JADE Administrator's Guide“ [9].

Das Ergebnis der Verknüpfung der Plattformen „mars“ und „venus“ aus Abbildung 2.2, lässt sich in Abbildung 2.3 erkennen. Zusätzlich zu den in Abbildung 2.2 dargestellten Agenten, existiert jeweils ein RMA Agent im Haupt-Container der Plattform. Dieser wurde gestartet, um das Ergebnis der Plattform-Verknüpfung grafisch darstellen zu können.

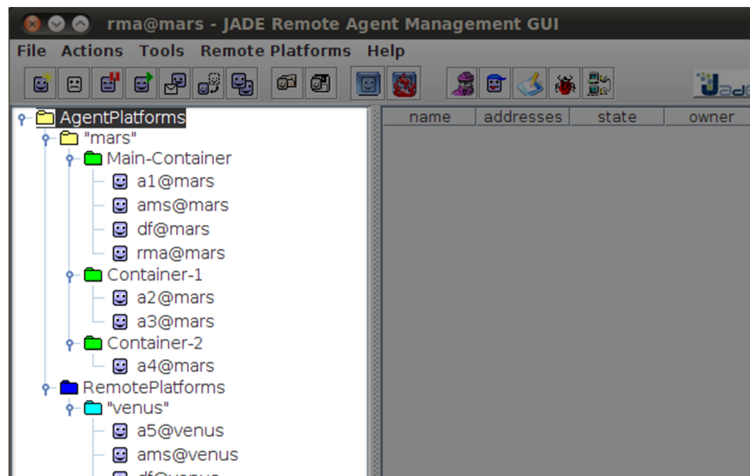

Abbildung 2.3: Der Haupt-Container von „mars“

2.3 Agenten in JADE

Ein Agent wird in JADE durch das Objekt einer Klasse abgebildet, welche von der Klasse „jade.core.Agent“ erbt (siehe Abbildung 2.4).

In der erbenden Klasse können die Methoden „setup“ und „takeDown“ überschrieben werden, um Ressourcen für den Agenten beim Start einzurichten (setup) und diese beim Beenden wieder freizugeben (takeDown). Die „receive“ Methoden bieten Zugriff auf die vom Agenten empfangenen Nachrichten. Über die „send“-Methode können Nachrichten an andere Agenten geschickt werden.

Zur Verwaltung einer Nachrichten-Warteschlange, besitzt jeder Agent ein „MessageQueue“-Objekt. Die „MessageQueue“ ist eine einfache Datenstruktur, die Methoden zum Hinzufügen („addFirst“ und „addLast“) und Entfernen („receive“) bereitstellt.

Die „receive“ Methoden der „Agent“-Klasse, arbeiten direkt mit der „receive“-Methode des „MessageQueue“-Klasse zusammen. Über die „receive“-Methode mit „MessageTemplate“-Parameter, lassen sich Nachrichten filtern. Während die einfache „receive“-Methode eine beliebige Nachricht aus der Warteschlange zurück liefert, lässt sich über das „MessageTemplate“ ein Muster definieren, auf das eine Nachricht passen muss, um aus der „MessageQueue“ zurückgeliefert zu werden. Die Verwendung dieser „receive“-Methode ist vor allem dann sinnvoll, wenn der Agenten auf eine Nachricht mit einer bestimmten performativen Äußerung (vgl. Abschnitt 1.3) wartet.

Das Verhalten eines Agenten, wird über eine Menge von Behaviour-Objekten gesteuert. In den Behaviour-Objekten, sind die eigentlichen Algorithmen des Agenten gekapselt. Ein Planer (Scheduler), wählt nach einer bestimmten Strategie eine der Behaviour-Objekte und bringt es zur Ausführung.

Der von JADE implementierte Scheduler arbeitet kooperativ. Im Gegensatz zu unterbrechenden Schedulern (präemptives Scheduling), wird eine Behaviour so lange ausgeführt, bis sie beendet ist (vgl. [10]). Dieses Verfahren bietet den Vorteil das zu einem Zeitpunkt immer genau eine Behaviour komplett ausgeführt wird. Durch die Verwendung eines kooperativen Schedulers, genügt ein Java-Thread pro Agent. Ein Entwickler kann so einen Agenten ohne zusätzliche Komplexität, welche durch Multi-Threading-Programmierung entstehen würde, entwickeln.

Möchte eine Entwickler die Mächtigkeit (und Probleme) mehrerer Threads verwenden, so können Behaviour-Objekte in einem eigenen Thread mit Hilfe der „ThreadedBehaviourFactory“ gestartet werden. Die konkrete Umsetzung kann im „JADE Programmer's Guide“ nachvollzogen werden [11].

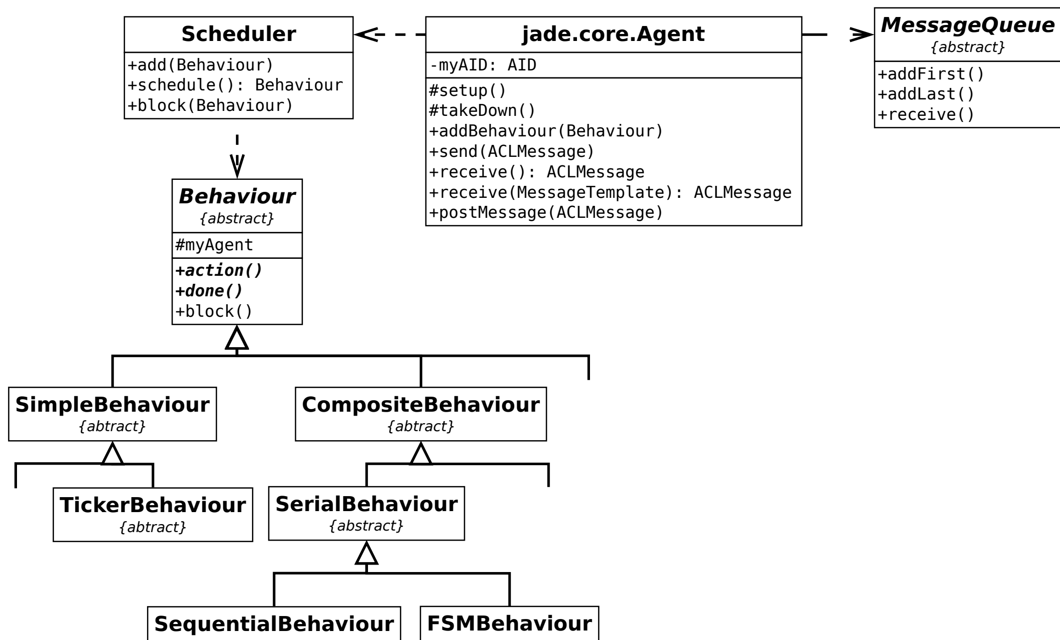

Abbildung 2.4: Ausschnitt aus dem UML-Diagramm für JADE Agenten

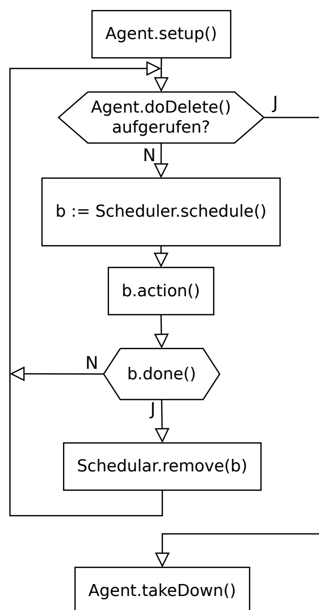

Abbildung 2.5: Ablauf eines Agenten

Das Ablaufdiagramm eines Agenten mit Scheduler und Behaviour-Objekten, wird in Abbildung 2.5 gezeigt. Beim Start eines Agenten wird die „setup"-Methode aufgerufen. Sie dient zum Einrichten des Agenten und zur Ressourcen-Allokation. Mindestens ein Behaviour-Objekt sollte hier dem Agenten hinzugefügt werden. Die „doDelete"-Methode kann jederzeit im Agenten-Objekt aufgerufen werden, um den Agent zu beenden. Wurde die Methode aufgerufen, so wird die „takeDown"-Methode ausgeführt. Die Haupt-Schleife des Agenten besteht aus einem Scheduler-Aufruf, der Ausführung der Behaviour und einem Test ob die Behaviour abgeschlossen ist.

Damit der Entwickler beim Entwickeln von Verhaltensklassen (Behaviour) unterstützt wird, wurden von JADE viele Anwendungsfälle generalisiert und in Form von Behaviour-Konkretisierungen bereitgestellt. In Abbildung 2.4 ist ein Ausschnitt aus der Klassenhierarchie unter Behaviour dargestellt. Die Blatt-Klassen der Hierarchiebaums aus erfüllen folgende Funktionen. Eine „TickerBehaviour" ist eine Behaviour, die in bestimmten Zeitabständen ausgeführt wird. Bei jder Ausführung wird die „onTick"-Methode aufgerufen.

Unter dem Zweig „CompositeBehaviour" sind Behaviour-Klassen angesiedelt, die sich aus weiteren Behaviour-Objekten zusammensetzten (vgl. Strukturmuster Kompositum [12]). So stellt ein sequenzielles Verhalten (SequentielBehaviour), die Ausführung seiner Behaviour-Objekte hintereinander bereit und wird beendet, nachdem das letzte Behaviour-Objekt abgearbeitet wurde.

Komplexere Anwendungsfälle können mit dem „Endlichen Automaten"-Verhalten (FSMBehaviour) modelliert werden. Die Ausführung dieses Verhaltens endet, wenn eine Verhalten durchlaufen wurde, welches einem Endzustand des Automaten entspricht.

Die hier gezeigten Klassen sind nur ein Ausschnitt. Ein kompletter Einblick ist über die JADE Application Programming Interface (API) Dokumentation im Paket „jade.core.behaviour" möglich [13].

2.4 Beispiel einer Verhandlung mit JADE

Um ein Beispiel für die konkrete Implementierung von Agenten zu geben, wird eine Verhandlung aus dem E-Commerce Umfeld in vereinfachter Form gezeigt. Der hier verwendete Ansatz orientiert sich am „JADE Programming For Beginners" Tutorial [8], verwendet jedoch an einigen Stellen unterschiedliche Ansätze [3].

Über ein Multiagenten-System werden Bücher verkauft. Es gibt zwei Klassen von Agenten: Händler und Kunden. Händler bieten eine Menge von Büchern zu einem festgelegten Preis an und Kunden möchten genau ein bestimmtes Buch kaufen. Wird ein Buch von mehreren Händler angeboten, so wird der Kunde das günstigste Angebot auswählen. Als Interaktionsprotokoll wird „FIPA Contract Net" eingesetzt (siehe 1.3). Die verwendete Ontologie ist in diesem Beispiel implizit und ergibt sich aus den verschickten Nachrichten. Ontologien werden im Abschnitt 2.5 genauer betrachtet.

2.4.1 Der Händler Agent

Der Quelltext des Händler-Agenten ist ausschnittsweise in Listing 2.4 dargestellt.

```
1 public class Haendler extends Agent {
2   protected void setup() {
3     processArguments();
4     if (registerDF()) {
5       addBehaviour(new SellOnContractNet(
6         this
```

[3] Der komplette Quelltext steht für Studenten und Mitarbeiter der FH Wedel unter `https://stud.fh-wedel.de/handout/Iwanowski/SeminarKI/` zur Verfügung. Alle Pfadangaben und Skripte beziehen sich auf Linux-Systeme.

```
7         , getContractNetMessageTemplate()));
8       } else {
9         doDelete();
10      }
11    }
12    protected void takeDown() {
13      deregisterDF();
14    }
15    // [...]
16  }
```

Listing 2.4: Ausschnitt aus dem Händler-Agenten

In Zeile 2 beginnt die „setup"-Methode, in der zuerst die Kommandozeilen-Parameter, durch den Aufruf von „processArguments" verarbeitet werden. Über diese Parameter bekommt der Händler-Agent die Bücher mit den Verkaufspreisen mitgeteilt. Es wäre denkbar das dies alternativ über eine Konfigurationsdatei oder über eine grafisches Benuzter-Schnittstelle konfiguriert wird. Der zweite Ansatz wird im „JADE Programming for Beginners" Tutorial verfolgt (siehe [8]).

Mit der Methode „registerDF" versucht sich der Agent in Zeile 4 am DF der Agenten-Plattform anzumelden, um von Kunden gefunden zu werden. Gelingt dies nicht, so kann dieser Agent keine Dienste anbieten und beendet sich selbst durch den doDelete Aufruf in Zeile 9.

Wurde der Agent am DF registriert, so erhält er sein Verhaltens-Objekt, mit dem die Verhandlung als Teilnehmer abgewickelt werden kann. Dieses Verhalten nutzt die von JADE bereitgestellte Behaviour-Klasse „ContractNetResponder" (siehe Listing 2.5).

```
1  private class SellOnContractNet
2      extends ContractNetResponder {
3
4     protected ACLMessage handleCfp(ACLMessage cfp)
5        throws RefuseException, FailureException, NotUnderstoodException { /* ... */ }
6
7      protected ACLMessage handleCfp(ACLMessage cfp)
8        throws RefuseException, FailureException, NotUnderstoodException { /* ... */ }
9
10     protected ACLMessage handleAcceptProposal(
11           ACLMessage cfp
12         , ACLMessage propose
13         , ACLMessage accept) throws FailureException { /* ... */ }
14 }
```

Listing 2.5: Behaviour des Händler-Agenten

Da der Händer-Agent nur die SellOnContractNet-Behaviour nutzt, die lediglich auf Auktionsnachrichten reagiert, ist der Agent rein reaktiv und besitzt keine proaktive Komponenten. Das bedeutet, dass der Händler-Agent seine Bücher niemals selbst anbieten wird, sondern nur auf Kaufanfragen reagieren kann.

Falls der Quelltext zu dieser Arbeit vorliegt (siehe Fußnote auf Seite 11), so kann im Verzeichnis „jade/seminar/demo/" mit dem Skript „02-start-haendler" ein neuer Container auf einer existierenden Plattform gestartet werden, welcher zwei Händler enthält.

2.4.2 Der Kunden Agent

Der Agent, der die Kunden-Rolle modelliert, funktioniert ähnlich wie der Händler-Agent (siehe Listing 2.6). In der „setup"-Methode wird zuerst versucht ein Kommandozeilen-Parameter zu extrahieren,

welches das gesuchte Buch spezifiziert. Gelingt dies nicht, so kann der Agent keine Aufgabe ausführen und beendet sich.

```
1  public class Kunde extends Agent {
2    protected void setup() {
3      if (!getBookArgument()) {
4        doDelete();
5      } else {
6        addBehaviour(new SearchForTraders());
7      }
8    }
9    protected void takeDown() {}
10   // [...]
11 }
```

Listing 2.6: Ausschnitt aus dem Kunden-Agenten

Im Unterschied zum Händler-Agenten, registriert sich der Kunde nicht mit dem DF, da er keine Dienste anbietet. Ein Kunde handelt ausschließlich proaktiv, indem er über seine einzige Behaviour „SearchForTraders“ nach Händlern sucht und bei Erfolg weitere Behaviour-Objekte dem Schedular hinzufügt (siehe Listing 2.7). Die Proaktivität des Agenten wird über die TickerBehaviour nachgebildet, die eine Ausführung der „onTick“ Methode alle 5 Sekunden veranlasst.

Wenn in der „SearchForTraders“-Behaviour die „onTick“-Methode aufgerufen wird, so wird eine Anfrage an der DF der Plattform gestellt, in der alle registrierten Händler-Agenten gesucht werden (Zeile 6). Wurden potentielle „Contract Net“ Auktionsteilnehmer gefunden, so wird durch das Hinzufügen der „BuyBookContractNet“-Behaviour eine neue Auktion initialisiert.

```
1  private class SearchForTraders extends TickerBehaviour {
2    public SearchForTraders() {
3      super(Kunde.this, 5 * 1000);
4    }
5    protected void onTick() {
6      if (getTradersFromDF()) {
7        myAgent.addBehaviour(new BuyBookContractNet(
8          Kunde.this
9        , getCFPMessage()));
10     }
11   }
12   // [...]
13 }
```

Listing 2.7: Behaviour des Kunden-Agenten

Zum Start der Auktion wird die von JADE bereitgestellt „ContractNetInitiator“-Behaviour genutzt. Die „Call-For-Propose“ Nachricht (siehe Beschreibung in Abschnitt 1.3), wird beim Start der Auktion an das „BuyBookContractNet“-Objekt übergeben und in der Methode „getCFPMessage“ erstellt (Zeile 9).

Falls der Quelltext zu dieser Arbeit vorliegt (siehe Fußnote auf Seite 11), so kann im Verzeichnis „jade/seminar/demo/“ mit dem Skript „03-start-kunde“ ein neuer Container auf einer existierenden Plattform gestartet werden. Der Container enthält ein Kunden, der ein Buch kaufen möchte, welches durch die Agenten angeboten werden, die mit dem Skript „02-start-haendler“ gestartet werden können [4].

[4] Um die Möglichkeit zu bieten, den Auktionsablauf mit Hilfe des Sniffer Agenten der JADE Plattform zu beobachten, wird der Kunde „schlafend“ (suspended) gestartet. Über die grafische Oberfläche (RMA) kann der Kunde „aufgeweckt“ (resume) werden.

2.4.3 Besonderheiten bei der Verarbeitung von Nachrichten

Eine wesentliche Aufgabe der Behaviour-Objekte ist die Verarbeitung von Nachrichten mit denen der Agent mit seiner Umwelt kommuniziert. Da mehrere Behaviour-Objekte innerhalb eines Agenten gestartet werden, sind unterschiedliche Nachrichten-Typen für unterschiedliche Behaviour-Objekte relevant.

```
1  public void action() {
2    MessageTemplate mt =
3      MessageTemplate
4        .MatchPerformative(ACLMessage.CFP);
5    ACLMessage msg = myAgent.receive(mt);
6    if (msg == null) {
7        block();
8    } else {
9      // weiterer Code
10   }
11 }
```

Listing 2.8: Verhaltensmuster zum Abruf einer Nachricht

Um ausschließlich relevante Nachrichten zu erhalten, rufen Behaviour-Objekte die „receive“-Methode mit „MessageTemplate„-Parameter auf, indem sie spezifizieren welche Nachricht erwartet wird. Liegt keine Nachricht vor, so liefert die „receive“-Methode „null“ zurück (siehe Listing 2.8). Wenn die „action“-Methode durchlaufen wird, obwohl keine neue Nachricht vorliegt, wird das Behaviour-Objekt mit der „block“-Methode, solange an der Ausführung gehindert, bis eine neue Nachricht vorliegt. Durch Anwendung dieses Verhaltensmusters, wird ein Busy-Waiting[5] vermieden

Die von JADE bereitgestellten Behaviour-Klassen berücksichtigen diesen Umstand bereits, weshalb dieses Muster in den Beispiel Agenten im Abschnitt 2.4.2 und 2.4.1 nicht implementiert werden musste.

2.5 Ontologien und Content-Sprachen in JADE

Die Ontologie im Beispiel aus Abschnitt 2.4 war aus vereinfachenden Gründen implizit über die Nachrichten-Typen definiert. JADE beherrscht ein komplexes Ontologie-System über das Ontologien definiert werden können, die zur semantischen Validierung von Nachrichten-Inhalten genutzt werden.

Eine Ontologie in JADE besteht aus folgenden Elementen:

- Konzept/Begriff (Concept)
- Agent-Aktion (AgentAction)
- Prädikat (Predicate)

Eine Agenten-Aktion ist ein spezielles „Konzept“ über das Aufgaben beschrieben werden. Die Elemente der Ontologie werden durch Java-Klassen repräsentiert, die jeweils das Interface des jeweiligen Ontologie-Elements implementieren. Die Vererbung bei „Konzepten“ wird direkt über das Java-Vererbung-Konzept realisiert.

```
1 class Gebaude     implements Concept  { /* ... */ }
2 class Hotel       extends    Gebaude  { /* ... */ }
3 class Person      implements Concept  { /* ... */ }
4 class Promienter  extends    Person   { /* ... */ }
```

[5] siehe http://de.wikipedia.org/wiki/Busy_Waiting

```
5 class Zimmer      implements Concept   { /* ... */ }
6 class WohntIn     implements Predicate { Person p; Gebauede g; /* ... */ }
7 class Vermietet   implements Predicate { Hotel h;  Zimmer z;   /* ... */ }
8
9 class HotelBuchen implements AgentAction { /* ... */ }
```

Listing 2.9: Hotel-Gast Ontologie von Seite 5 in JADE

In Listing 2.9 sind die Ontologie-Elemente aus der Beispiel Ontologie, die im Abschnitt 1.4 (Seite 5) definiert wird, skizziert. Es ist zu erkennen das die Instanzen von Prädikaten über Felder in den Klassen verwaltet werden. Als Beispiel für eine Agenten-Aktion wurde zusätzlich die „HotelBuchen“-Aktion hinzugefügt, die von einem Agenten genutzt werden kann um eine „Person“ auf ein bestimmtes „Hotel“ zu buchen.

Damit JADE mit der Ontologie arbeiten kann, müssen die einzelnen Ontologie-Elemente zu einer Ontologie vereint werden. Zu diesem Zweck existiert die „BeanOntology“-Klasse, die beerbt werden kann. Durch Nutzung des „Konvention über Konfiguration“-Ansatzes wird dem Entwickler viel Konfigurationsarbeit erspart. Es genügt im Konstruktor der abgeleiteten Klasse alle Ontologie-Elemente durch die „add“-Methode hinzuzufügen, um die Ontologie-Klasse zu definieren. Die „add“-Methode erkennt über die Getter- und Setter-Methoden der Klasse die internen Felder und erstellt entsprechende Ontologie-Schemata.

```
1  class HotelGastOntologie extends BeanOntology {
2    /* ... */
3    public EmploymentOntology(String name)
4        throws BeanOntologyException {
5      add(Gebaude.class);
6      add(Hotel.class);
7      add(Person.class);
8      add(Promienter.class);
9      add(Zimmer.class);
10     add(WohntIn.class);
11     add(Vermietet.class);
12   }
13 }
```

Listing 2.10: Erstellen der Ontologie mit Hilfe der BeanOntology-Klasse

In Listing 2.10 wird die Hotel-Gast-Ontologie mit Hilfe der „BeanOntology“-Klasse erstellt. Näheres zum Thema Ontologie mit der „BeanOntology“-Klasse wird im JADE Tutorial „Creating ontologies by means of the bean-ontology class“ [14] beschrieben. Falls der Quelltext zu dieser Arbeit vorliegt (siehe Fußnote auf Seite 11), kann die Ontologie im Verzeichnis „jade/seminar/src/fhw/seminar/onto/“ im Detail betrachtet werden.

Um die semantisch spezifizierten Nachrichten-Inhalte in einer konkreten Nachricht zu transportieren, wird eine Syntax benötigt. JADE stellt zu diesem Zweck Content-Sprachen zur Verfügung.

Der Ablauf bei der Transformation von Ontologien zur Nachrichten und zurück ist in Abbildung 2.6 dargestellt. Der Prozess der Konvertierung umfasst die Umwandlung in eine Zwischenrepräsentation - das abstrakte Content-Element (AbsContentElement). Für den Entwickler ist dieser Prozess jedoch transparent durch einen „ContentManager“ gekapselt.

Ein Ablauf der Nachrichten-Erstellung und -Extrahierung wird in Listing 2.11 dargestellt.

In Zeile 3 und 5 werden Content-Sprache und Ontologie beim ContentManager des Agenten registriert. Ab Zeile 8 wird eine Nachricht mit der performativen Äußerung „inform“ erstellt[6]. Es wird

[6] ACLMessage ist die Klasse für alle Agenten-Nachrichten. ACL steht für Agent Communication Language

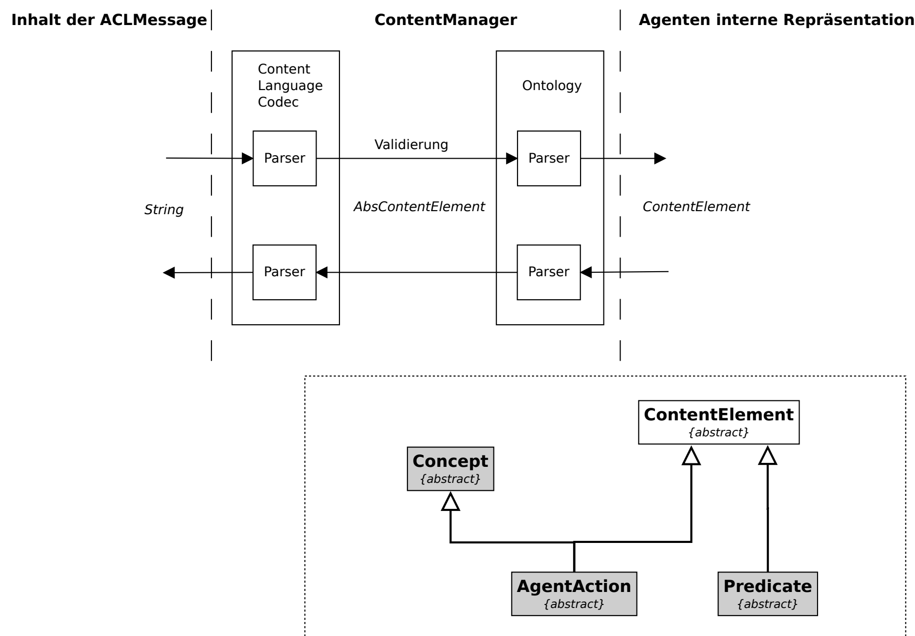

Abbildung 2.6: Die Content-Sprachen Ontologie Pipeline

festgelegt in welcher Ontologie der Nachrichten-Inhalt formuliert ist und mit welcher Content-Sprache der Inhalt kodiert wird. Der Nachrichten-Inhalt wird in Zeile 11 vom ContentManager erstellt.

```
1  // Ontologie und ContentLanguageCodec registrieren
2  Codec codec = new SLCodec();
3  getContentManager().registerLanguage(codec);
4  Ontology ontology = MyOntology().getInstance();
5  getContentManager().registerOntology(ontology);
6
7  // Nachricht versenden
8  ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
9  msg.setLanguage(codec.getName());
10 msg.setOntology(ontology.getName());
11 getContentManager().setContent(msg, myContentElement);
12
13 // Nachricht empfangen
14 ContentElement ce = getContentManager().extractContent(msg);
```

Listing 2.11: Registrierung von Content-Sprache und Ontologie im ContentManager und Nutzung des ContentManagers

Die Erstellung vom Content-Element „myContentElement“ ist kein Bestandteil dieses Beispiels. Ein Content-Element ist entweder ein Prädikat oder eine Agenten-Aktion (wie in Abbildung 2.6 dargestellt). Die Festlegung eines Konzeptes (Concept) als Inhalt einer Nachricht wird somit ausgeschlossen.

Die Umwandlung einer empfangenen Nachricht ist in Zeile 14 skizziert. Falls der Inhalt einer Nachricht syntaktisch oder semantisch fehlerhaft ist, werden von ContentManager entsprechende Exceptions ausgelöst. Eine detaillierte Betrachtung des ContentManager bietet das JADE Tutorial „Creating and using applications-specific ontologies“ [15].

Content-Sprachen können, neben der bereits erwähnten Funktionalität, optional eine innere Ontologie zur Verfügung stellen. Ziel einer inneren Ontologie ist die Verknüpfung von Prädikaten und

Aktionen zu komplexen Termen.

Die von der FIPA spezifzierte Content-Sprache „Semantic Language (SL)“ (siehe [16]) stellt über eine innere Ontologie folgende Möglichkeiten zur Verfügung:

- Logische Operationen: AND, OR, NOT
- Identifying Referential Expression (IRE): ANY, ALL, IOTA
- Angaben zum Zustand von Fakten: BELIEF, UNCERTAIN
- Alternative-Aktionen, Sequenzen von Aktionen

Über eine IRE lassen sich Instanzen anhand von Prädikaten auswählen. Ein Beispiel für die IRE „IOTA“ in SL-Syntax ist in Listing 2.12 dargestellt und beschreibt die Frage nach der Hauptstadt von Deutschland.

```
1 ((iota ?x (hauptstadt deutschland ?x)))
```

Listing 2.12: Eine IRE in SL-Syntax

„IOTA“ referenziert dabei immer genau eine Instanz. Würde ein Agent in seiner Wissensbasis mehrere Stadt-Instanzen kennen, für die das Prädikat „hauptstadt“ mit dem Parameter „deutschland“ war ist, so müsste dieser ein leeres Resultat auf die „IOTA“-Frage liefern.

Eine detaillierte Beschreibung von SL findet sich in der FIPA Spezifikation [16].

2.6 Weitere Technologien

Bei der Recherche zu dieser Arbeit wurde weitere interessante Agenten-Technologien gefunden, die als Anknüpfungspunkte kurz erwähnt werden sollen.

Die Universität Hamburg entwickelt mit Jadex[7] ein Belief Desire Intention (BDI) Open-Source Framework für die Entwicklung von Agenten. Details zu der BDI Architektur bietet der Seminar-Vortrag von Kai Hasselbach [17]. Agenten die mit Jadex entwickelt werden, sind aufgrund der FIPA Kompatibilität von Jadex auch auf der JADE Plattform lauffähig. Der Einstieg in die Entwicklung mit Jadex wird durch viele quelloffene Beispielprogramme erleichtert.

Die Modellierung von unterschiedlichen Aufgabenstellung wird durch die NetLogo[8] Umgebung mit Hilfe von Agenten ermöglicht. Eine gute Dokumentation und eine sehr große Sammlung bereits modellierter Aufgabenstellung helfen beim Entwickeln mit NetLogo.

[7] http://jadex-agents.informatik.uni-hamburg.de/
[8] http://ccl.northwestern.edu/netlogo/

3 Einschätzung und Bewertung

3.1 Schwächen von JADE

Einige Schwächen des JADE Frameworks sollen nicht unerwähnt bleiben.

Einer der Hauptnachteile fällt beim Blick auf den Quelltext und die API von JADE auf. Da die Code-Basis zwischen 2000 und 2001 entwickelt wurde, muss an vielen Stellen mit typunsicheren Downcasts gearbeitet werden. Viele Schnittstellen liefern beispielsweise „Object“-Objekte als Ergebniss. Auch Generics[1] werden weitgehend nicht verwendet, da diese erst seit der Java Version 1.5 unterstützt werden, die 2004 erschienen ist.

Ein weiterer Nachteil der Plattform ist das Fehlen von Sicherheitsfeatures, die auch von der FIPA nicht standardisiert wurden. So kann die Authentizität von Agenten-Nachrichten nicht geprüft werden und der Inhalt von Nachrichten wird grundsätzlich im Klartext verschickt. Technisch ist es möglich eine Authentifizierung und Verschlüsselung zu implementieren, nur fehlt unter Umständen die Kompatibilität zu Agenten andere Anbieter. JADE hat diesen Nachteil erkannt und versucht ihn durch die Addons „Security“ und „Trusted Agents“[2] zu beheben. Diese Addons wurden von der JADE-Community entwickelt und im Rahmen dieser Arbeit nicht weiter betrachtet.

Durch die Integration der mobilen Java-Version (J2ME) ist an vielen Stellen im JADE Quelltext eine bedingte Compilierung notwendig, welche die Lesbarkeit und Wartbarkeit des Quelltextes einschränkt. Werden Agenten entwickelt, die auch auf J2ME lauffähig sein sollen, so ist eine Anwendung dieser Technik zu prüfen.

3.2 Stärken von JADE

Der Hauptvorteil von JADE liegt in der standardkonformen Implementierung der FIPA-Spezifikationen, so dass eine Zusammenarbeit mit Agenten anderer Anbieter und anderen Plattformen sichergestellt ist.

Durch die Implementierung der „FIPA Agent Management Specification“ [7] ist das Framework gut skalierbar und lässt sich aufgrund der Java-Technologie auf vielen Plattformen ausliefern.

JADE kann weiterhin voll in eigene Produkte integriert werden, da eine Nutzung der „jade.Boot“-Klasse zum Starten der Plattform nicht gefordert wird. So kann vorhandene Software leicht um Agenten-Fähigkeiten erweitert werden.

Die Weiterentwicklung von JADE wird durch das „JADE Board“ vorangetrieben, in dem sich Unternehmen wie „Motorola“ oder die „Telecom Italia“ befinden. Das die Koordinierung durch das „JADE Board“ funktioniert, zeigt sich auch dadurch, dass die letzte JADE Version, zum Zeitpunkt dieser Arbeit, vor 2 Monaten, im April 2010, erschienen ist.

Der Einstieg in die Entwicklung mit JADE wird durch viele Tutorials und Beispielprogramme erleichtert. Alle Konzepte können im JADE Quelltext nachvollzogen werden, da dieser vollständig zur Verfügung steht.

[1] `http://de.wikipedia.org/wiki/Generics`

[2] Die JADE Addons finden sich unter `http://jade.tilab.com/community-addons.php`

3.3 Multiagenten-Systeme

Der Einsatz von Multiagenten-Systemen eignet sich nach Meinung des Autors für größere Softwaresysteme die folgende Eigenschaften erfüllen.

Zum einen sollte das System eine lose Kopplung der Komponenten erfordern, die durch Multiagenten-Systeme optimal abgebildet werden kann. Sind zum anderen viele unterschiedliche Organisationen an der Entwicklung des Systems beteiligt, so wird durch eine klare Ontologie-Spezifikation die Entwicklung des Systems vereinfacht, wenn diese durch Agenten abgebildet werden kann.

Das Gebiet der Multiagenten-System ist auch weiterhin ein interessantes Forschungsgebiet, welches durch eine weltweit aktive Gemeinschaft vorangetrieben wird. Auf der „Autonomous Agents and Multiagent Systems (AAMAS)“ Konferenz, welche 2010 in Torronto (Kanada) stattfand, werden in regelmäßigen Abständen neue Forschungsergebnisse präsentiert. Auch die FIPA ist auf diesen Konferenzen vertreten, so dass eine weitere Standardisierung im Agenten-Umfeld durch die FIPA zu erwarten ist.

Literaturverzeichnis

[1] Wooldridge, Michael: *An Introduction to MultiAgent Systems.* Wiley, 2009.

[2] Foundation for Intelligen Physical Agents: *FIPA Communicative Act Library Specification.* http://www.fipa.org/specs/fipa00037/, 2002. [Online; abgerufen am 27. Mai 2010].

[3] Austin, John Langshaw: *How to Do Things with Words.* Harvard University Press, 1962.

[4] Wikipedia: *Sprechakttheorie — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Sprechakttheorie&oldid=75228931, 2010. [Online; abgerufen am 6. Juni 2010].

[5] Foundation for Intelligen Physical Agents: *FIPA Contract Net Interaction Protocol Specification.* http://fipa.org/specs/fipa00029/, 2002. [Online; abgerufen am 27. Mai 2010].

[6] Wikipedia: *Ontologie (Informatik) — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Ontologie_%28Informatik%29&oldid=75327778, 2010. [Online; abgerufen am 10. Juni 2010].

[7] Foundation for Intelligen Physical Agents: *FIPA Agent Management Specification.* http://fipa.org/specs/fipa00023/, 2002. [Online; abgerufen am 31. Mai 2010].

[8] Caire, Giovanni: *JADE PROGRAMMING FOR BEGINNERS.* http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf, 2009. [Online; abgerufen am 29. April 2010].

[9] Bellifemine, Fabio et al.: *JADE Administrator's Guide.* http://jade.tilab.com/doc/administratorsguide.pdf, 2010. [Online; abgerufen am 22. April 2010].

[10] Wikipedia: *Scheduling — Wikipedia, Die freie Enzyklopädie*, 2010. [Online; abgerufen am 12. Juni 2010].

[11] Bellifemine, Fabio et al.: *JADE Programmer's Guide.* http://jade.tilab.com/doc/programmersguide.pdf, 2010. [Online; abgerufen am 22. April 2010].

[12] Schmidt, Prof. Dr. Uwe: *Softwaredesign: Kompositum.* http://www.fh-wedel.de/~si/vorlesungen/softwaredesign/Strukturmuster/Kompositum.html, 2010. [Online; abgerufen am 12. Juni 2010].

[13] *JADE API.* http://jade.tilab.com/doc/api/index.html, 2010. [Online; abgerufen am 12. Juni 2010].

[14] Cancedda, Paolo und Giovanni Caire: *Creating ontologies by means of the BeanOntology class.* http://jade.tilab.com/doc/tutorials/BeanOntologyTutorial.pdf, 2010. [Online; abgerufen am 13. Juni 2010].

[15] Caire, Giovanni und David Cabanillas: *Creating and using applications-specific ontologies*. http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf, 2010. [Online; abgerufen am 13. Juni 2010].

[16] Foundation for Intelligen Physical Agents: *FIPA SL Content Language Specification*. http://fipa.org/specs/fipa00008/, 2002. [Online; abgerufen am 13. Juni 2010].

[17] Hasselbach, Kai: *Software-Agenten*. http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI/Ausarbeitung12AgentenHassel.pdf, 2005. [Online; abgerufen am 13. Juni 2010].