Jan Sudeikat

Betrachtung und Auswahl der Methoden zur Entwicklung von Agentensystemen

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Softwaretechnik
am Fachbereich Elektrotechnik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Christoph Klauck
Zweitgutachter : Prof. Dr. Winfried Lamersdorf
[Universität Hamburg, Fachbereich Informatik]

Abgegeben am 26 Mai 2004

Jan Sudeikat

Thema der Diplomarbeit

Betrachtung und Auswahl der Methoden zur Entwicklung von Agentensystemen

Stichworte

Softwareagenten, Multiagentensystem, agentenorientierte Methoden, agentenorientierte Entwicklung, agentenorientiertes Software–Engineering

Kurzzusammenfassung

Methoden des Software–Engineering unterstützen die systematische, planmäßige und zielgerichtete Erstellung qualitativ hochwertiger Software. Die Entwicklung von Multiagentensystemen ist eine junge Disziplin in der Informatik. In den letzten Jahren wuchs in diesem Bereich das Bewusstsein, dass Methoden notwendig sind, um Entwicklung von hochqualitativen Multiagentensystemen zu ermöglichen.
In dieser Diplomarbeit werden Methoden zur Entwicklung dieser Systeme betrachtet. Ein Überblick, über vorgeschlagene Methoden, wird gegeben und deren Anwendbarkeit für eine bestimmte Plattform wird untersucht.
Bei dieser Plattform zur Entwicklung von Agentensystemen handelt es sich um das Jadex System. Diese Erweiterung der JADE Agenten Plattform wird am Arbeitsbereich Verteilte Systeme & Informationssysteme (VSIS) des Fachbereichs Informatik der Universität Hamburg entwickelt.

Jan Sudeikat

Title of the paper

Consideration and Selection of Methodologies for the Development of Agent Systems

Keywords

Software agents, multi–agent system, agent–oriented methodologies, agent–oriented development, agent–oriented software–engineering

Abstract

Methodologies in software–engineering provide means for the systematic, planned and purposeful development of high-quality software. The development of multi agent systems is a new discipline in computer science. In recent years the research community got aware of the need for methodologies, to allow development of high quality multi–agent systems.
In this diploma thesis methodologies for the development of these systems are considered. An overview of suggested methodologies is given and their applicability for a concrete platform is examined.
This platform for the development of agent systems is the Jadex System. This add–on to the JADE agent platform is developed at the Distributed and Information Systems unit (VSIS) in the Department of Computer Science of Hamburg University.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Agentenorientierung, als junges Forschungsfeld der Informatik, schlägt ein neues Paradigma vor, um die steigende Komplexität moderner Softwaresysteme beherrschbar zu machen. Hiernach dient die Metapher eines *Agenten*, eines autonomen Vertreters von Interessen, als grundlegendes Mittel der Modellierung und Programmierung von Anwendungen. Nachdem Architekturen und Plattformen zur Anwendung dieses neuen Ansatzes entwickelt wurden, erhält dieser Zweig der Forschung zunehmend mehr Aufmerksamkeit. Im Zuge dessen Verbreitung, wächst auch das Bewusstsein für die Notwendigkeit des Einsatzes von *Methoden* für die strukturierte, planmäßige und zielgerichtete Entwicklung von Anwendungen nach diesem neuen Ansatz.
So wurden in den letzten Jahren einige Hilfsmittel zur agentenorientierten Anwendungsentwicklung vorgeschlagen. Diese Vorschläge reichen von Anpassungen bekannter Techniken des Entwurfes von Softwaresystemen, bis hin zu kompletten Methoden mit eigenen Vorgehensmodellen und Notationen. Bisher ließen deren Gegenüberstellungen in der Fachliteratur die zur Anwendung kommenden Plattformen außer acht.

Diese verschiedenen Plattformen bestimmen die Implementation der einzelnen Agenten. Verschiedene Architekturen haben sich als zweckdienlich herausgestellt. Interdisziplinäre Überlegungen führten zu dem Ansatz, mentalistische Konzepte in Agenten zu integrieren. Eine konkrete Plattform zur Entwicklung agentenorientierter Anwendungen, wurde am Arbeitsbereich *Verteilte Systeme & Informationssysteme* (VSIS) des Fachbereichs Informatik der Universität Hamburg entwickelt. Das *Jadex* System erweitert eine verbreitete Open Source Agentenplattform um solche Konzepte.

Aufgabenstellung

In diesem Zusammenhang verfolgt die vorliegende Diplomarbeit zwei Ziele. In erster Linie sollen Methoden zur Entwicklung von Agentensystemen im Hinblick auf eine konkrete Plattform untersucht werden. Es wird untersucht inwieweit die Methoden geeignet sind die Entwicklung von Anwendungen, unter Verwendung der Jadex Plattform, zu unterstützen. Zusätzlich ist die Integration bereits bestehender Werkzeuge einer geeigneten Methode, in die Anwendungsentwicklung wünschenswert. Hierzu wird mittels eines Prototypen untersucht, in welchem Umfang Design–Artefakte in Modelle der Plattform übertragen werden können.

Vorgehensweise

Um diese Aufgaben zu lösen, wird zunächst ein Überblick über die vorgeschlagenen Methoden zur agentenorientierten Anwendungsentwicklung gewonnen. Es werden die Methoden identifiziert, die die Entwicklung von Agenten mir den oben genannten mentalistischen Konzepten (an–)leiten. Nach einer Einschätzung der Möglichkeiten dieser Ansätze, einer Betrachtung der Eigenschaften des Jadex Systems und einer Analyse der bisher erschienenen Arbeiten zum Vergleich von Methoden des Software–Engineering im Allgemeinen und agentenorientierter Methoden im Besonderen, wird ein Katalog von Bewertungskriterien erstellt. Es werden Kriterien extrahiert, die für die Benutzung der betrachteten Plattform relevant sind.
Die ausgewählten Methoden werden, mittels dieser Kriterien, einander gegenüber gestellt. Um hierfür einen angemessenen Eindruck von deren Benutzbarkeit zu bekommen, wird ein Fallbeispiel mit den in Frage kommenden Exemplaren modelliert. Diese Überlegungen führen zur Auswahl der geeignetsten Methode.
Für die Entwicklung des oben genannten Prototypen ist ein eingehender Vergleich, zwischen den in den Design–Artefakten enthaltenen Informationen und denen mittels der Jadex–Agenten beschrieben werden, nötig. Um die dessen Benutzbarkeit zu bewerten, wird eine Beispielanwendung entwickelt. An diesem Beispiel wird der Weg von der Modellierung eines Systems bis zur fertigen Codierung gezeigt und erprobt.

Aufbau der Arbeit

Im folgenden Kapitel wird ein Überblick über die verwendeten Begriffe gegeben. Es wird geklärt was in dieser Arbeit unter Agenten, Multiagentensystemen, etc. verstanden wird.
Daraufhin werden im dritten Kapitel die Grundlagen der agentenorientierten Softwareentwicklung vorgestellt. Der Sinn und Zweck von Methoden des Software–Engineering im Allgemeinen wird gezeigt, bevor der Bedarf an agentenbasierten Methoden erläutert wird. Anschließend wird ein Überblick über bereits vorgeschlagene Methoden zur Entwicklung von Agentensystemen gegeben.
Um eine Bewertung einzelner Entwicklungsmethoden einzuleiten, wird im vierten Kapitel ein Überblick über die Arten gegeben, mit denen bisher Methoden miteinander verglichen wurden, und welche Arbeiten bereits agentenorientierte Methoden betrachteten. Aus diesen Arbeiten wird eine Menge von Bewertungskriterien extrahiert, die die Anwendbarkeit für ein konkretes System beeinflussen.
Es folgt die entsprechende Bewertung einzelner Methoden im fünften Kapitel.
Im sechsten Kapitel werden dann drei der betrachteten Methoden in Analyse und Design eines Fallbeispieles eingesetzt. Die gesammelten Erfahrungen, zusammen mit den Ergebnissen bereits im vierten Kapitel vorgestellter Arbeiten, führen zu einer abschließenden Bewertung.
Das siebte Kapitel beschreibt dann einen Prototypen, der im Zuge dieser Arbeit entwickelt wurde. Aus Designartefakten einer Methode werden für die Plattform lesbare Beschreibungen einzelner Agenten generiert. Die Entwicklung (bis hin zur Implementation) eines Fallbeispiels mit Hilfe dieses Werkzeugs wird beschrieben und führt zu einer Betrachtung der Anwendbarkeit desselben. Die Grenzen der Anwendbarkeit der Methode und der Codegenerierung werden aufgezeigt.
Diese Arbeit endet in einer kurzen Zusammenfassung der Ergebnisse, und einem Ausblick. In diesem werden Anregungen für weiterführende Untersuchungen gegeben.

Die zu Rate gezogene Literatur war fast ausschließlich auf Englisch. Soweit sich in der deutschen Fachliteratur Begriffe eingebürgert haben, werden diese benutzt. Ansonsten wurde auf eine Übersetzung verzichtet, um keine unnötige Verwirrung zu stiften. Wenn es der Verständlichkeit zu dienen schien, wurden Übersetzungen einzelner Begriffe in Klammern angegeben.
Die in dieser Arbeit aufgeführten Referenzen auf Internetseiten wurden zuletzt am 13. April 2004 überprüft.

Diese Diplomarbeit entstand im Rahmen einer Kooperation zwischen der Hochschule für Angewandte Wissenschaften (HAW) Hamburg und dem Arbeitsbereich *Verteilte Systeme & Informationssysteme* (VSIS) des Fachbereichs Informatik der Universität Hamburg. Betreuende Personen waren Prof. Dr. Klauck (HAW), Prof. Dr. Lamersdorf (VSIS) und die beiden Entwickler der Jadex Plattform Lars Braubach (VSIS) und Alexander Pokahr (VSIS).

Eine verkürzte Darstellung, der in dieser Diplomarbeit gefundenen Ergebnisse, wird in einem Beitrag zum *Fifth International Workshop on AGENT-ORIENTED SOFTWARE ENGINEERING*[1] (AOSE-2004) veröffentlicht (Sudeikat et al., 2004).

[1] http://www.jamesodell.com/aose2004

Kapitel 2

Agentensysteme

In diesem Kapitel werden im Weiteren verwendete Begriffe, Konzepte und Softwaresysteme vorgestellt. Insbesondere der Begriff der Agenten wird interdisziplinär verwendet, was eine Eingrenzung für den Kontext dieser Arbeit nötig macht.

2.1 Agenten – Versuch einer Definition

Der Begriff *Agent* entstand aus interdisziplinären Überlegungen und insbesondere die englische Sprache verwendet dieses Wort in einer Vielzahl von Begriffen (*estate agent*, *forwarding agent*, *insurance agent*, *sole agent*, etc.). Dies scheint dazu zu verleiten, diesen Begriff zur Beschreibung verschiedenster Softwaresysteme zu benutzen.
In einer frühen Arbeit haben Franklin und Graesser (1996) den Begriff des Agenten eingegrenzt. Sie erstellten hierzu die in Abbildung 2.1 dargestellte Taxonomie. Sie unterscheiden die Softwareagenten von Lebewesen (*Biological Agents*), von Robotern (*Robotic Agents*) und von der Forschungsrichtung des Künstlichen Lebens (*Artificial Life*). Letztere simuliert Lebenstrukturen und -prinzipien durch Populationen autonomer Programme (beschrieben von Langton, 1988).
Van Steen und Tanenbaum (2002:173) definieren die dort eingeordneten *Software*

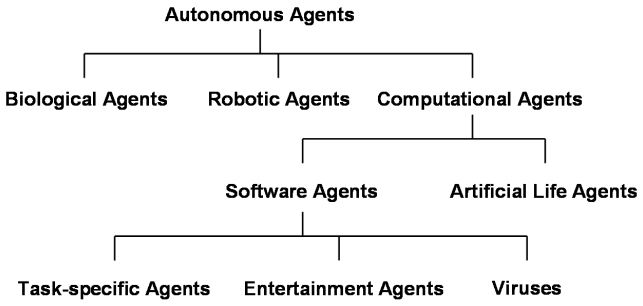


Abbildung 2.1: Taxonomie von (Software–)Agenten nach Franklin und Graesser (1996)

Agents als:

> ... we define a **software agent** as an autonomous process capable of reacting to, and initiating changes in, its environment possibly in collaboration with users and other agents. The feature that makes an agent more then just a process is its capability to act on its own, and, in particular, to take initiative where appropriate.

Sie verstehen Agenten also als spezielle Prozesse in Verteilten Systemen, die sich dadurch auszeichnen autonom zu agieren und Initiative zu zeigen. Diese Beschreibung ist vage, stellt aber einen 'kleinsten gemeinsamen Nenner' dar, der die in dieser Arbeit betrachteten Agentenarten beschreibt.

Um die Anforderungen an Agenten die miteinander kooperieren zu spezifizieren, unterschieden zuerst Wooldridge und Jennings (1995:115) zwischen einer starken und schwachen Auffassung von Agenten(*strong / weak notion of agency*). Agenten beider Arten zeichnen sich durch folgende Merkmale aus.

Schwache Agenten (weak notion of agency) sind:

autonom: sie arbeiten unabhängig und haben vollständige Kontrolle über ihre Aktionen und ihren internen Zustand.

sozial: sie interagieren mit anderen Agenten.

reaktiv: sie nehmen ihre Umwelt (und sei es nur über Nachrichten anderer Agenten) wahr und reagieren darauf.

pro–aktiv: sie können initiativ zielorientiertes Verhalten zeigen, um ihre Umwelt zu beeinflussen.

Starke Agenten (strong notion of agency) verwenden meist mentalistische Konzepte. Sie erweitern die obigen Eigenschaften um:

Mobilität: Sie können sich in der realen Welt (Roboter) oder in einem Netzwerk bewegen.

Aufrichtigkeit: Agenten werden nicht bewusst falsche Informationen weitergeben.

Wohlwollen: Die Ziele der Agenten widersprechen sich nicht, Agenten versuchen immer, die ihnen zugewiesenen Aufträge auszuführen.

Rationalität: Agenten werden sich (im Rahmen ihrer Möglichkeiten) immer so verhalten, dass ihre Ziele erfüllbar bleiben.

Während die schwache Auffassung auf viele Programme zutrifft (beispielsweise auch auf Betriebssystem–Prozesse), wird die starke Auffassung vor allem in Arbeiten der (Verteilten) Künstlichen Intelligenz vertreten. Eine weitere Eigenschaft, die Agenten (insbesondere im diesem Bereich) oft zugeschrieben wird, ist die Fähigkeit, zu lernen (beispielsweise bei Nwana, 1996).

2.2 Wie unterscheiden sich Agenten von Objekten?

Oft wird das Konzept der autonomen Agenten als eine natürliche Weiterentwicklung des objektorientierten Paradigma vorgestellt. Wie Jennings (2001:39) darstellt, beruhen sowohl Objekte als auch Agenten auf Interaktionen (mittels ausgetauschter Nachrichten) und bedienen sich des *Geheimnisprinzips* (Information Hiding).
Bezeichnenderweise wird im Sprachgebrauch dazu geneigt, Objekte zu vermenschlichen, was indirekt anzeigt, dass sie als Abstraktionen für aktive Artefakte einer modellierten Realität stehen. Auch die objektorientierte Analyse kommt ohne die Beschreibung von aktiven Elementen nicht aus. Booch (1994) schreibt:

> ... we view the world as a set of autonomous agents that collaborate to perform some higher level function ...

Wooldridge und Ciancarini (2001) finden drei grundlegende Unterschiede zwischen Agenten und Objekten. Objekte kapseln ihr Verhalten und die in ihnen enthaltenen Daten. Dadurch verbleibt die Kontrolle über die Daten beim Objekt. Methoden eines Objektes werden von anderen Objekten *aufgerufen*. Ein Objekt hat keine Kontrolle über sein Verhalten, eine aufgerufene Methode muss ausgeführt werden. Agenten hingegen stellen und reagieren auf *Anfragen* zur Ausführung gewisser Aktivitäten. Die Entscheidung, ob einer Anfrage entsprochen wird, verbleibt beim einzelnen Agenten.
Als zweiten Unterschied identifizieren sie die Möglichkeit, mit Hilfe von Agenten flexibles, autonomes Verhalten (reaktiv, pro–aktiv, sozial) zu modellieren. Diese Art von Verhalten kann zwar in objektorientierten Systemen nachgebildet werden, wird aber in deren Modellen nicht explizit unterstützt.
Zuletzt beschreiben sie die Agenten innewohnende Nebenläufigkeit. Sie sind immer aktiv, während Objekte meist zwischen einzelnen Methodenaufrufen keine Verarbeitungen durchführen. Zwar enthalten die objektorientierten Programmiersprachen Möglichkeiten, Nebenläufigkeit zu programmieren und sie werden genutzt, um Agenten (und Agentensysteme) zu implementieren. Trotzdem wird dieses Konzept der Agenten als autonome Einheiten nicht unterstützt.

Jennings (2001) betont in diesem Zusammenhang, dass die objektorientierte Sicht Strukturen komplexer Systeme, zu feinkörnig darstellt. Er bezieht sich dabei auf die drei grundlegenden Prinzipien, die Booch (1994) gefunden hat, um die in Softwaresystemen inhärente Komplexität beherrschbar machen.

Dekompositon Die Aufteilung eines Problems in kleine, überschaubare Einheiten

Abstraktion Die Erstellung von vereinfachenden Modellen, in denen relevante Eigenschaften hervorgehoben und andere vernachlässigt werden.

Hierarchisierung Erstellung hierarchischer Strukturen, in denen Systeme in abstraktere zusammengefasst werden.

Jennings (ebd.) argumentiert, wie Agenten als Design–Metapher diese Prinzipien auf effektive Weise unterstützen. Eine Modellierung in (Organisationen von) Agenten erlaubt die Modularisierung des Systems nach den Aufgaben die sie erfüllen. Außerdem muss für ein System aus autonomen Einheiten nicht während des Designs über Art und Umfang der Kommunikation entschieden werden. Ein solches

Design stellt eine äußerst komplexe Aufgabe dar. Stattdessen entscheiden Agenten zur Laufzeit über Art und Umfang der benötigten Kommunikation.
Nach Jennings bietet es sich an, Agenten als Abstraktion zu verwenden, da Teilsysteme naturgetreu als Organisationen von Agenten abgebildet werden. Sie bestehen aus einer Menge von Komponenten, die passend auf Rollen agieren und interagieren. Die Interaktionen lassen sich leicht zu sozialen Interaktionen (zwischen Agenten und Organisationen) abstrahieren.
Die entstehende Hierarchie aus Organsiationen, Teilorganisationen und Agenten kann auf allen Ebenen als eine Sammlung atomarer Einheiten betrachtet werden. Da diese Einheiten autonom agieren und auf einer semantischen Ebene miteinander kommunizieren, können sie weitgehend unabhängig voneinander entwickelt werden.

Kritiker wenden ein, dass die beschriebenen agentenorientierten Konzepte mit objektorientierten realisiert werden können, aber der Wert des neuen Paradigma liegt darin, Entwicklern eine passende Metapher und geeignete Techniken an die Hand zu geben (Jennings, 2001:41).

2.3 Architekturen von Agenten

Grundsätzlich wird zwischen zwei Ausprägungen von Agenten unterschieden: den *reaktiven* und den *deliberativen* Agenten (auch kognitive Agenten genannt).

Reaktive Agenten reagieren reflexartig auf äußere Einflüsse, wohingegen deliberative Agenten auf der *physical symbol hypothesis* von Newell und Simon (1976) beruhen. Sie verfügen über ein symbolisches Modell ihrer Umwelt. Durch Operationen auf den Symbolen und Suchen in Mengen der Symbole, kann intelligentes Verhalten erzielt werden.

Die *Subsumption–Architektur* ist ein prominentes Beispiel einer Architektur reaktiver Agenten. Brooks (1990; auch von Nielsen (1998:32) beschrieben) hat sie erfolgreich in zahlreichen Roboter–Systemen eingesetzt. Das Verhalten dieser Roboter wird durch verschiedene *behaviour modules* gesteuert. Sie alle erhalten direkt die Sensordaten des Roboters. Sobald diese Daten bestimmte Vorbedingungen erfüllen, veranlassen die Module entsprechende Aktionen. Den einzelnen Modulen sind nach Prioritäten geordnet, die bestimmen welche der veranlassten Aktionen letztendlich ausgeführt werden. Mittels so einfacher Konstruktionen konnten erstaunlich komplexe Verhaltensweisen erzeugt werden.

Eine der bekanntesten Architekturen für deliberative Agenten ist die sog. *BDI–Architektur*. Bratman (1987) entwickelte ein Modell, in dem rationales Verhalten, durch die Konzepte *Belief* (Überzeugungen), *Desire* (Wünsche) und *Intention* (Absichten) beschrieben wird.

Beliefs stellen die grundlegenden Ansichten und Erwartungen eines Agenten dar.

Desires beschreiben Überzeugungen, die erfüllt werden sollen.

Intentions sind die (Handlungs–)Absichten eines Agenten, um einzelne Ziele zu erfüllen.

Die Beliefs repräsentieren das *Wissen* eines Agenten. Agenten nehmen ihre Umwelt war, das gewonnene Wissen wird in den Beliefs gespeichert. Wie der Begriff schon andeutet, kann (und wird wahrscheinlich) dieses Wissen nicht immer der realen Umwelt entsprechen. Es handelt sich vielmehr um den *Eindruck* den Agenten von ihrer Umwelt haben.
Rao und Georgeff (1995) entwickelten hieraus eine formale Theorie und ein ausführbares Modell. Sie führten die konkreteren Konzepte der *Goals* (Ziele) und *Plans* (Pläne) ein. Ziele repräsentieren die Wünsche, die ein Agent zu erfüllen versucht. Diese werden üblicherweise durch Zustände in den Beliefs dargestellt. Pläne beinhalten mehrere Intentionen (und evtl. untergeordnete Teilziele), um ein Ziel zu erreichen. Sie enthalten die Anweisungen, mit denen der Agent einzelne Ziele erreichen kann. Folglich besteht ein ausführbares BDI–Modell lediglich aus den Beliefs, Zielen und Plänen. Diese Architektur hat sich bei der Entwicklung von Agenten mit mentalen Zuständen durchgesetzt (Wooldridge und Ciancarini, 2001).

Das *Procedural Reasoning System* (PRS) war das bekannteste System, dass diese (konkretisierte) BDI–Architektur implementierte (Georgeff und Lansky, 1987). Manchmal wird der Begriff PRS synonym für diese Art von BDI–Architekturen benutzt. Es wurde ursprünglich am Stanford Research Institute (später am Australian AI Institute (AAII)) entwickelt und in einer Reihe von Anwendungen eingesetzt. Wooldridge und Ciancarini (2001) beschreiben dieses System als sehr solide und erprobt.

Capabilities wurden von Busetta et al. (1999) als Module zur Organisation des internen Aufbaus von BDI–Agenten vorgestellt. Sie werden definiert als eine Sammlung von Plänen, Beliefs und Events, sowie Regeln, die die Sichtbarkeit dieser drei Komponenten nach außen beschreiben.

Diese oben vorgestellten Architekturen stellen Extreme dar. Es wurden eine Reihe von *hybriden Architekturen* vorgeschlagen, die reaktive und deliberative Ansätze miteinander vereinen. PRS-artige Systeme werden zu diesen hybriden Architekturen gezählt (Mangina, 2002:9), da sie oft auch in der Lage sind, reaktives Verhalten zu zeigen. Es sind Konfigurationen möglich, in denen ein Ereignis (z. B. eine empfangene Nachricht) die Erzeugung spezieller Ziele anstößt, deren einzige Aufgabe die Ausführung eines einzelnen Planes ist.

2.4 Multiagentensysteme

Wie in Abschnitt 2.1 dargestellt wurde, ist ein zentrales Merkmal der Agenten ihre Zusammenarbeit. Ein System aus kooperierenden Agenten wird als Multiagentensystem (MAS) bezeichnet. Ferber (1995) beschreibt Agenten und Agentensysteme vor allem aus dem Blickwinkel der *Verteilten Künstlichen Intelligenz*. Er definiert ein MAS als System, das aus den folgenden Elementen besteht:

- Einer Umwelt E.
- Eine Menge von situierten Objekten O. Situiert bedeutet hier, dass zu jedem Zeitpunkt den Objekten eine Position zugeordnet ist. Die Agenten können Ob-

jekte wahrnehmen, erzeugen, modifizieren und löschen. Die Kommunikation zwischen den Agenten erfolgt über den Austausch von Nachrichten (ebenfalls Objekte).

- Eine Menge von Agenten *A*. Sie werden als aktive Objekte angesehen ($A \subseteq O$).
- Eine Menge von Beziehungen *R* verbindet die Objekte miteinander.
- Eine Menge von Operatoren *Op*, mit denen Agenten Objekte empfangen, erzeugen, konsumieren, verändern und löschen können. Zusätzlich gibt es Operatoren, die die Veränderungsversuche an Objekten und die Reaktionen der Umwelt auf diese darstellen.

Ausgehend von diesem Modell können verschiedene Ausprägungen von MAS beschrieben werden. *Situierte Agentensysteme* interagieren mit der oben erwähnten Umwelt. Als *rein situierte Agentensysteme* werden Systeme beschrieben, in denen die beteiligten Agenten nicht unmittelbar über Nachrichten miteinander kommunizieren, sondern sich nur über die jeweils verursachten Veränderungen in der Umwelt wahrnehmen.
Werden Nachrichten zur Kommunikation verwendet, wird das System als *kommunizierendes Multiagentensystem* bezeichnet. Als *rein kommunizierendes Multiagentensystem* wird ein System bezeichnet, in dem die Umwelt nur aus Agenten besteht ($E = O \wedge A = O$), und die Agenten (nur) über Nachrichten miteinander kommunizieren. Diese Art von Systemen wird häufig in der Verteilten Künstlichen Intelligenz verwendet. In dieser Arbeit wird nur die Entwicklung dieser Systeme betrachtet. Ferber (1995) benutzt den Begriff des Softwareagenten synonym für rein kommunizierende Agenten.

Agentenplattformen

Als Agentenplattformen (*agent platform*) wird eine spezielle Art von Middleware bezeichnet (Van Steen und Tanenbaum, 2002:175), die grundlegende Dienste zur Realisierung von Multiagentensystemen zur Verfügung stellt. Nachdem Plattformen unabhängig voneinander entwickelt wurden, haben sowohl die *Object Management Group*[1] (OMG) als auch die *Foundation for Intelligent Physical Agents*[2] (FIPA) unabhängig von einander verschiedene Standards entwickelt.
Die von der OMG definierte *Mobile Agent Facility* (MAF) verwendet die *Common Object Request Broker Architecture*[3] (CORBA; ebenfalls von der OMG spezifiziert) zur Kommunikation und zum Transport der Agenten. Es handelt sich bei CORBA um eine objektorientierte Middleware, die plattformübergreifende Protokolle und Dienste definiert. Ziel dieser Architektur sind verteilte Anwendungen in heterogenen Umgebungen. Der MAF–Standard definiert eine generelle Architektur und eine eigene Schnittstelle in der von CORBA verwendeten *Interface Definiton Language*[4] (IDL), die kompatible Plattformen implementieren müssen (Architektur und Schnittstelle werden von der Object Management Group, 2000 beschrieben). Dieser Ansatz wurde nie verwirklicht.

[1] http://www.omg.org
[2] http://www.fipa.org
[3] http://www.omg.org/gettingstarted/corbafaq.htm
[4] http://www.omg.org/gettingstarted/omg_idl.htm

Es gibt einige Plattformen, die die FIPA-Spezifikationen implementieren (Architektur beschrieben in: Foundation for Intelligent Physical Agents, 2002). Grundsätzlich bieten Plattformen dieser Architektur Möglichkeiten, Agenten zu erzeugen und zu löschen, andere Agenten zu lokalisieren und Kommunikationsmöglichkeiten zwischen den Agenten. Ein generelles Modell wird in Abbildung 2.2 dargestellt. Über

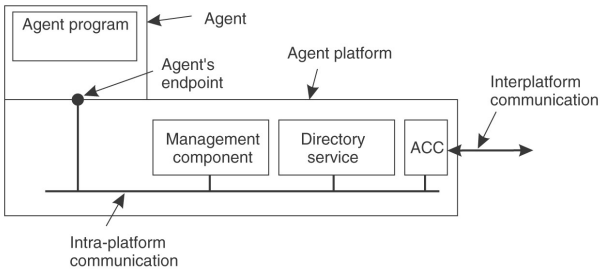


Abbildung 2.2: FIPA - Agentenplattform (van Steen und Tanenbaum, 2002)

eine Schnittstelle benutzen Agenten die Funktionen der Plattform. Die *Management component* verwaltet die Agenten die über *Agent endpoints* mit der jeweiligen Plattform interagieren.
Der *local directory service* (auch oft Directory Facilitator (DF) genannt) kann von entfernten Plattformen aus angesprochen werden, um einzelne Agenten zu erreichen. Hier werden die *Services* registriert, die die jeweiligen Agenten anbieten. Sie sind analog zu den Operationen in der objektorientierten Programmierung zu sehen. Agenten können über den DF angebotene Services identifizieren und ihre Ausführung *anfragen*. Sie werden nur angefragt und nicht aufgerufen, da der Agent die Kontrolle behält. Aufgrund seines inneren Zustandes kann er sich gegen eine Ausführung des Service entscheiden und diese ablehnen.
Die Kommunikation zwischen den Agenten erfolgt über den *agent communication channel* (ACC). Der ACC ermöglicht eine zuverlässige Punkt–zu–Punkt Kommunikation für Agenten auf verschiedenen Plattformen. Die Agenten kommunizieren miteinander über Nachrichten. Der Austausch von Nachrichten erfolgt auf Anwendungsebene mittels Kommunikationsprotokollen. Diese werden als *agent communication language* (ACL) bezeichnet. Unterschieden wird zwischen dem Zweck einer Nachricht und seinem Inhalt. Das Format der Nachrichten und die möglichen Zwecke der Nachrichten sind in weiteren FIPA Spezifikationen definiert. Tabelle 2.1 zeigt ein Beispiel für eine ACL–Nachricht. In diesem Beispiel teilt ein Agent einem anderen Informationen über eine Musikgruppe mit. Die Informationen werden in Prolog übertragen, und es wird eine Ontologie zur Interpretation der Informationen angegeben.

2.5 Die JADE Plattform

JADE (Java Agent DEvelopment Framework) ist ein in Java™implementiertes Framework des Telecom Italia Lab[5] (TILab). Es ermöglicht die Implementation von

[5]http://www.telecomitalialab.com/index_e.htm

Feld	Wert
Performative	INFORM
Sender	max@toolband.fanclub.org:7815
Receiver	peter@http://music_magazine.com:5623
Language	Prolog
Ontology	Rockbands
Content	band(Tool) singer(Maynard Keenan, Tool)

Tabelle 2.1: Beispiel einer FIPA ACL Nachricht

Multiagentensystemen durch eine FIPA konforme Middleware, eine Klassenbibiothek und Werkzeuge, um die Entwicklung zu unterstützen. Zur Kommunikation zwischen den Agenten kann Java™RMI, IIOP und HTTP benutzt werden. Weitere Transportmechanismen sind geplant. Diese Plattform wird unter unter der LGPL–Lizenz (Lesser General Public License Version 2) veröffentlicht und von mehreren Firmen und akademischen Institutionen eingesetzt[6].

2.5.1 Der Aufbau

Die in Abschnitt 2.4 vorgestellten Management Component und Directory Service werden in JADE als Agenten realisiert. Diese werden automatisch beim Start einer Plattform instanziiert. Agenten werden innerhalb von Laufzeitumgebungen, den sog. *Containern*, ausgeführt. Für jede Plattform existiert mindestens einer (der sog. maincontainer) in dem mindestens die beiden genannten Agenten ausgeführt werden. Benutzer können eine beliebige Anzahl von neuen Containern instanziieren. Eine Plattform kann auch über mehrere Hosts verteilt ausgeführt werden. Zur internen Kommunikation wird RMI verwendet. Abbildung 2.3 zeigt eine solche Plattform, die über drei Rechner verteilt ist. Die Container der Plattform werden innerhalb der Java Runtime Environment™(JRE) eines jeden Hosts ausgeführt. Innerhalb dieser Container werden wiederum die einzelnen Agenten ausgeführt.

2.5.2 Programmierung von Agenten

Agenten werden in JADE als einzelner Thread ausgeführt. Die jeweiligen Aufgaben eines Agenten werden in sog. *Behaviours* implementiert. Sie bestimmen die Aktionen, die die einzelnen Agenten ausführen und werden von einem Scheduler nach dem Round–Robin Verfahren nicht preemptiv zur Ausführung zugelassen. Die einzelnen Behaviour–Objekte sind dafür zuständig, die Ausführungskontrolle an Andere abzugeben, ein Multitasking im eigentlichen Sinne wird nicht unterstützt.
Zusätzlich existiert eine ausführliche Klassenbibliothek, die zahlreiche von der FIPA spezifizierte Standards implementiert, um sie Anwendungsprogrammierern zur Verfügung zu stellen. Dazu gehört die Benutzung von FIPA ACL–Nachrichten, Ontologien und Interaktionsprotokollen. Eine umfassende Einführung in die Anwendungsentwicklung mit JADE wird von Bellifemine et al. (2003) gegeben.

[6]Liste der Anwender und weitere Informationen unter: http://jade.cselt.it/

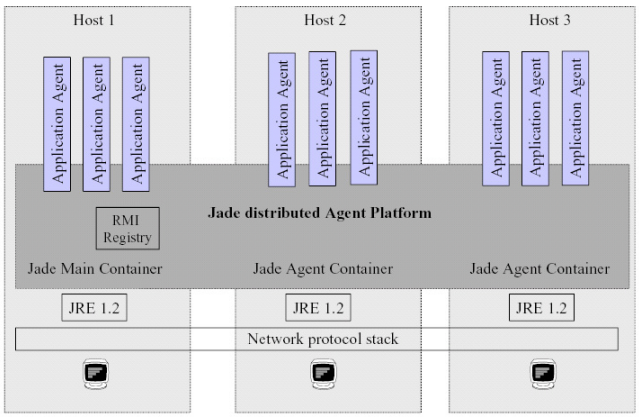


Abbildung 2.3: Eine Jade-Plattform über mehrere Hosts verteilt (aus Bellifemine et al., 2003)

2.5.3 Mitgelieferte Werkzeuge

Das JADE–System stellt einige Werkzeuge zur Verfügung, um die Administration einzelner Agentenplattformen und das Testen von Agenten zu erleichtern.
Die *DFGUI* stellt grafisch den DF einer Plattform dar. Die Inhalte des DF können verändert und Verbindungen zu anderen DF aufgebaut werden. Alle weiteren Werkzeuge wurden als eigenständige Agenten implementiert. Der *Remote Monitoring Agent* (RMA) stellt ein grafisches Werkzeug zur Administration einer Agentenplattform dar. Er wird automatisch bei der Instanziierung einer Agentenplattform gestartet, gibt einen Überblick über die auf einem Rechner vorhandenen Laufzeitumgebungen und die in ihnen laufenden Agenten. Es können neue Agenten und Laufzeitumgebungen gestartet und auch entfernte Umgebungen administriert werden. Von diesem RMA aus kann der *Introspector* gestartet werden. Mit ihm ist es möglich, einen laufenden Agenten sowie dessen Nachrichtenaustausch zu überwachen und zu kontrollieren. Die ausführbaren Behaviours können eingesehen und schrittweise ausgeführt werden. Der *Dummy Agent* stellt eine grafische Schnittstelle zur Verfügung, mittels derer ACL–Nachrichten an beliebige andere Agenten gesandt werden können. Auch die Details von gesendeten und empfangenen Nachrichten werden angezeigt. Dies soll das Testen und Debuggen von Agenten unterstützen. Auch der sog. *Sniffer* wurde als eigener Agent implementiert. Er ermöglicht die Beobachtung und grafische Darstellung des Nachrichtenaustausches zwischen Agenten. Der *Socket Proxy Agent* kann von Agenten benutzt werden, um mit entfernten Anwendungen über Sockets zu kommunizieren. Dieser Agent empfängt Nachrichten auf einem anzugebenden Port, sucht in diesen Nachrichten nach dem Namen des Agenten, für den die Nachricht bestimmt ist und leitet sie dann an diesen weiter.

2.6 Die Jadex Erweiterung

Das Jadex System erweitert die JADE Plattform um die oben beschriebene BDI–Architektur. Seit der Version 0.9 (Oktober 2003) ist Jadex als Erweiterung von JADE erhältlich. Dieser Abschnitt beschreibt den Aufbau und die ereignisorientier-

te Arbeitsweise der BDI–Agenten (Detaillierte Beschreibungen sind bei Pokahr und Braubach (2003) und Pokahr, Braubach und Lamersdorf (2003) zu finden).

2.6.1 Definition der Agenten

Wie Abbildung 2.4 zeigt, besteht die Definition eines Jadex–Agenten aus zwei Teilen. Soll ein Agenten erzeugt und gestartet werden, müssen seine Eigenschaften dem System bekannt gemacht werden. Dies geschieht über XML-Dateien, die sog. *Agent Definition Files* (ADF). Sie beschreiben die dem Agenten bekannten Pläne, seine initialen Ziele und Beliefs und angebotenen Services (letztere, damit diese dem DF bekannt gemacht werden können). Eine detaillierte Beschreibung des Aufbaus und Inhaltes dieser Dateien wird später in Abschnitt 7.1.1 gegeben. Neben dieser deklarativen Beschreibung stehen den Agenten ausführbare Pläne zu Verfügung. Der ADF eines Agenten referenziert diese, sie werden in separaten Java™–Klassen ausprogrammiert. Auch das Konzept der Capabilities (siehe 2.3), zur Modularisierung

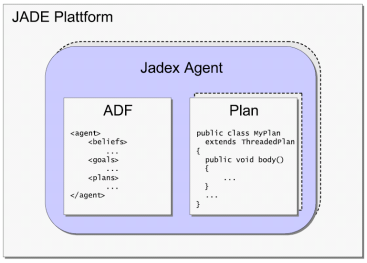


Abbildung 2.4: Jadex - Bestandteile eines Agenten (aus Pokahr und Braubach, 2003)

von Agenten, wird in Jadex verwendet. Die ADF eines Agenten kann weitere ADFs referenzieren, die einzelne Capabilities beschreiben.

2.6.2 Die BDI–Architektur in Jadex

In Jadex wird die oben beschriebene BDI–Architektur mit den Mitteln des JADE Frameworks realisiert. Daher lassen sich dessen Klassen und Werkzeuge zur Entwicklung von Jadex–Agenten weiterverwenden. In der *Belief Base* eines Agenten kann jedes Java™–Objekt (unter einem Schlüssel, *identifier* genannt) gespeichert werden. Diese Werte (Objekte) der gespeicherten Beliefs werden *facts* genannt. Zwei Arten der Speicherung werden unterstützt. Ein *belief* speichert ein einzelnes Objekt, ein *beliefset* speichert eine Menge von Objekten unter einem Schlüssel. Eine deklarative Sprache (ähnlich der Object Query Language (OQL)) kann verwendet werden, um die gespeicherten Daten abzufragen. Zu einem oder mehreren Beliefs können über die Erfüllung von Bedingungen (*conditions*) Ereignisse ausgelöst werden.
Goals werden in der *goal base* gespeichert, *achieve goals* beschreiben die zu erreichenden Zustände. Diese Zustände sollten einen gemeinsamen, konsistenten Zustand beschreiben. Das System überprüft nicht, ob Ziele miteinander vereinbar sind. So ist es möglich, dass Pläne sich gegenseitig die Erfüllung von Zielen verhindern.

Das Jadex System kennt auch noch zwei andere Arten von Zielen. Ein *maintain goal* wacht über die Erfüllung einer Bedingung (*maintainconditon*). Ist diese Bedingung nicht mehr erfüllt, wird das Ziel aktiviert. Ein *perform goal* spezifiziert direkt, dass bestimmte Aktionen ausgeführt werden sollen.

Um die in den Goals beschriebenen Zustände herbeizuführen, werden die Pläne ausgeführt. Die Definition eines Planes besteht aus einem sog. *head*, der den Plan im ADF beschreibt und dem *body*, der Java™–Klasse, die die Funktionalität des Planes implementiert. Plans werden instanziiert, um auf Ereignisse zu reagieren, oder Ziele zu erfüllen. Im Head können sog. *filter* angegeben werden, um die Ereignisse zu spezifizieren. Bei der Erzeugung eines Agenten werden seine *instant plans* ausgeführt. Es gibt zwei Ausführungsarten für Pläne. Sie können entweder als eigener Thread ausgeführt werden, was bedeutet, dass sie blockierend auf Ereignisse warten, ohne den gesamten Agenten zu beeinträchtigen oder kompatibel zu den Behaviours in JADE sein, was kein blockierendes Warten ermöglicht (Details hierzu sind im *JADE Programmer's Guide* (Bellifemine et al., 2000–2003) zu finden).
Abbildung 2.5 zeigt die konzeptionelle Architektur eines solchen Agenten. Ein Agent reagiert auf eingehende Nachrichten (Messages) und interne Ereignisse (Internal Events; meint innerhalb des Agenten). Außerdem versucht er seine Ziele (adopted Goals) zu erfüllen. Die internen Events können von Plänen oder durch die Belief Base (die Erfüllung von Bedingungen triggern Ereignisse) ausgelöst werden. Um Ziele zu erfüllen oder auf Ereignisse zu reagieren, können (instanziierte) Pläne ausgeführt werden. Diese können wiederum Nachrichten senden, das Wissen des Agenten (Belief Base) verändern, neue Goals zur Erfüllung erstellen und interne Ereignisse generieren.

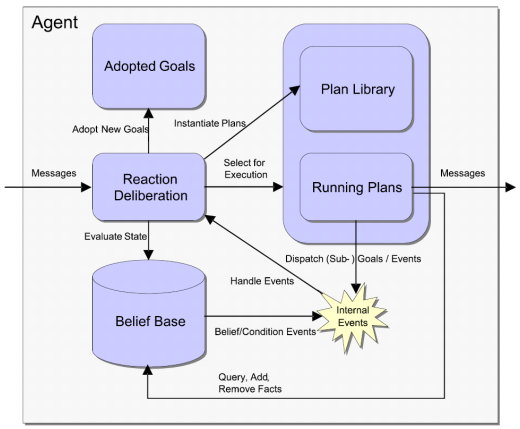


Abbildung 2.5: Jadex - Architektur eines Agenten (aus Pokahr und Braubach, 2003)

2.6.3 Die Ausführung der Agenten

Die Agenten bestehen aus mehreren internen Datenstrukturen und vier Prozessen. Diese Prozesse werden als Behaviours des JADE–Systems ausgeführt (im Round–

Robin–Verfahren) und haben gemeinsamen Zugriff auf die Datenstrukturen. Abbildung 2.6 zeigt die Prozesse (Ovale) und die Datenstrukturen (Rechtecke). Die

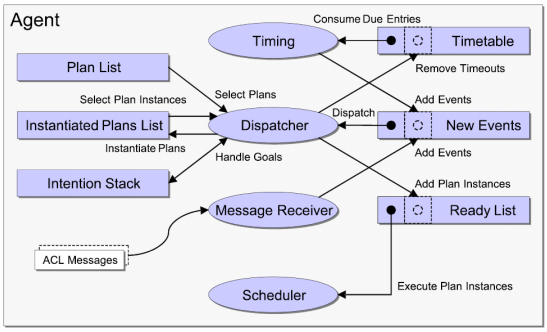


Abbildung 2.6: Jadex - Ausführung eines Agenten (aus Pokahr und Braubach, 2003)

Prozesse *message receiver* und *timing process* fügen lediglich neue Ereignisse in die Liste *New Events* ein. Der Message Receiver wartet auf den Empfang von ACL–Nachrichten und erzeugt entsprechend neue Events. Die *Timetable* ist eine geordnete Liste von Ereignissen, die zu einem bestimmten Zeitpunkt verarbeitet werden sollen. Der Timing Process entnimmt diese der Timetable und fügt sie den neuen Ereignissen hinzu.
Die *Ready List* enthält Instanzen von Plänen, die bereit sind ausgeführt zu werden. Der *Scheduler*–Prozess ist dafür zuständig, diese Pläne nacheinander, schrittweise auszuführen. Dazu wird der erste Plan der Liste entnommen, ein Schritt ausgeführt und die mit dem Plan assoziierten Goals entsprechend angepasst. Wurde der letzte Schritt eines der Pläne erreicht, wird er aus der Liste gelöscht. Die *Plan List* enthält alle dem Agenten bekannten Pläne. Der Dispatcher kann diese inztanziieren, diese Instanzen werden dann in der *Instantiated Plan List* gespeichert. Der *Intention Stack* enthält alle Goals des Agenten.
Der *Dispatcher*–Prozess entnimmt das nächste zu verarbeitende Ereignis aus der Liste der New Events und ist dafür zuständig, einen passenden Plan für dessen Verarbeitung zu finden. Für entnommene Ereignisse wird geprüft, ob ein Goal mit ihm assoziiert ist und eine Liste der anwendbaren Pläne wird erstellt. Sind diesem Goals zugeordnet, entscheidet das sog. *Meta–Level Reasoning* (das vom Anwendungsprogrammierer erweitert werden kann) darüber, welcher Plan anzuwenden ist. Sind keine Goals assoziiert, wird das Ereignis an alle anwendbaren Pläne weitergereicht. Wurde ein Goal nicht erreicht, ist der Dispatcher dafür zuständig, einen weiteren Plan zu seiner Erfüllung auszuführen.

2.6.4 Mitgelieferte Werkzeuge

Es existiert ein eigener *Jadex RMA*, der den RMA des JADE–Systems um eigene Funktionen erweitert (siehe Pokahr und Braubach, 2003). Beim Start eines Agenten kann ein ADF angegeben werden, und zwei eigene Werkzeuge können gestartet werden.

Das erste Werkzeug ist der *Jadex Introspector*. Ähnlich dem JADE Introspector können hier die internen Abläufe in den Agenten beobachtet werden. Es existieren zwei Ansichten. Im *BDI Viewer* werden die Beliefs, Goals und Plans eines Agenten angezeigt. In der *Intospector*–Ansicht kann die Verarbeitung der Events, ähnlich eines Debuggers, überwacht und schrittweise ausgeführt werden.

Außerdem kann ein *Logger Agent* gestartet werden. Er erlaubt die Ansicht von Logging–Nachrichten, die in den Plans anderer Agenten generiert werden. Programmierer der Plans generieren die angezeigten Nachrichten in ähnlicher Weise, wie in den verschiedenen Logging Frameworks, die für die Sprache Java™ erhältlich sind (Java™ Logging API[7], Log4J[8], etc.).

[7] http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/
[8] http://logging.apache.org/log4j/docs/

Kapitel 3

Agentenorientierte Softwareentwicklung

In diesem Kapitel wird der Bedarf an agentenorientierten Entwicklungsmethoden dargestellt, und einen Überblick über vorgeschlagene Methoden geben.

3.1 Methoden der Software–Technik

Die Software–Technik ist eine Teildisziplin der Praktischen Informatik. Balzert (2000) definiert Software–Technik als:

> Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software–Systemen. Zielorientiert bedeutet die Berücksichtigung z.B. von Kosten, Zeit, Qualität.

Arbeitsteilig und umfangreich meint hier die Koordination von mehreren Entwicklern bei der Entwicklung über einen längeren Zeitraum. Ingenieursmäßig ist die Entwicklung durch ihre Marktorientierung. Es gilt den optimalen Kompromiss zwischen der Qualität des Produktes, der Zeit und den Kosten zur Entwicklung zu finden.
Unter den Prinzipen werden Grundsätze verstanden, die dem Handeln zugrunde gelegt werden, beispielsweise Abstraktion, Hierarchisierung und Modularisierung. In dieser Arbeit werden die in der Definition genannten Methoden und Werkzeuge betrachtet.

In der englischen Literatur zur Softwarentwicklung wird unterschieden zwischen *methods* und *methodologies* (Booch, 1994). In der deutschen Literatur werden diese Begriffe als Verfahren oder (Kern–)Arbeitsschritte (für methods) und Methoden (für methodologies) übersetzt.

Aufgrund der fast ausschließlich auf Englisch gehaltenen Literatur zur agentenorientierten Softwareentwicklung und dem Gleichklang dieser englischen und deutschen Begriffe, kann diese Übersetzung auf den ersten Blick verwirren. Deutschen Autoren fällt oft die Unterscheidung zwischen der methodology und der Methodologie

schwer. Das deutsche Wort "Methodologie" bedeutet aber so viel wie: *Methodenlehre*, im Sinne der Lehre von den Wegen wissenschaftlicher Erkenntnis.
Die Unterscheidung zwischen den Begriffen ist also wichtig. Daher werden im Weiteren die Begriffe, die sich in der deutschen Literatur zur Softwaretechnik durchgesetzt haben, verwendet.

Methoden wurden von Rumbaugh et al. (1991) definiert als:

> a process for the organiszed production of software, using a collection of predefined techniques and notational conventions. A methodology ... is usually presented as a series of steps, with techniques and notation associated with each step.

Diese Definition führt drei Bestandteile ein. Eine *Vorgehensweise* als eine Sammlung von Arbeitsschritten, verbunden mit einzelnen *Techniken* und *Notationen*, die in den einzelnen Schritten zu verwenden sind.
Abbildung 3.1 zeigt eine Übersicht dieser drei Bestandteile aus einer Arbeit von Sturm und Shehory (2003). Unter der Sammlung von verwendeten Techniken werden hier Metriken (Metrics), Qualitätssicherung (Quality Assurance QA), die Einhaltung von Standards (zur Codierung, Kommunikation, etc.) und die Verwendung von Werkzeugen (Tools) gefasst. Diese Werkzeuge unterstützen hierbei die verwendete Notation (Modellierungssprache, Modeling Language). Die Vorgehensweise teilt nicht nur die Entwicklung der Software in einzelne Arbeitsschritte auf (Procedure), sondern unterstützt auch das Projektmanagement (Project Management), teilt den Mitarbeitern in einem Projekt fest definierte Rollen (Roles) zu, die bestimmen, welche Aufgaben sie während der Entwicklung zu erfüllen haben und legen fest, wie die Modellierung der Software unter Verwendung der Notationen dokumentiert wird (Deliverable).
Keine der bisher vorgeschlagenen Methoden zu agentenorientierten Entwicklung unterstützt alle dieser Komponenten. Dies ist vor allem darauf zurückzuführen, dass nur wenige Entwicklungserfahrungen mit diesen Methoden gesammelt wurden. Nur mit einem Schatz an Erfahrungen können Metriken und (Qualitäts–)Standards verifiziert werden. Das Management eines agentenorientierten Softwareprojektes (verbunden mit der Verteilung der Rollen) wird kaum adressiert. Im besten Fall geben Methoden Heuristiken zum Entwurf vor.

3.2 Bedarf an agentenorientierten Entwicklungsmethoden

Agentensysteme sind nicht weit verbreitet. Einige Studien (Jennings und Wooldridge, 2000; Jennings, 2000; Cernuzzi und Rossi, 2002; Dam und Winikoff, 2003) machen hierfür neben dem meist experimentellen Stadium der Agentenplattformen, hauptsächlich den Mangel an ausgewachsenen, erprobten Methoden verantwortlich. Insbesondere kommerzielle Entwickler könnten sich an diesen orientieren, erst mit solchen Hilfsmitteln werden agentenorientierte Systeme in der Industrie als Entwicklungsansatz verwendbar sein (zitiert nach Iglesias, Garijo und González, 1999).
In einer Sammlung von Richtlinien und Empfehlungen der europäischen Forschungsinitiative AgentLink[1] wird von Luck, McBurney und Preist (2003:4) ebenfalls der

[1] http://www.agentlink.org

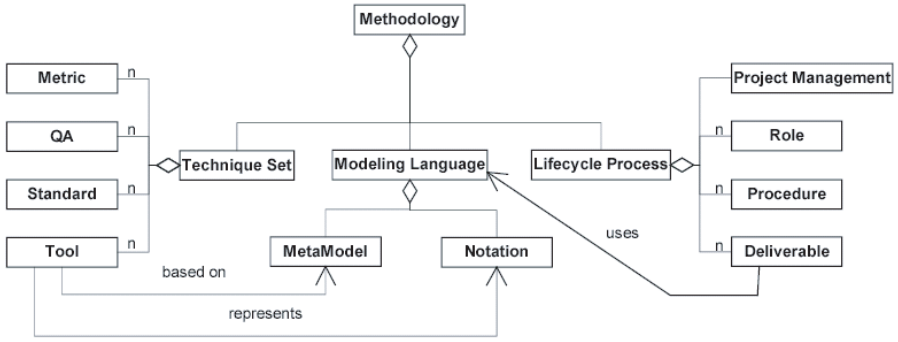


Abbildung 3.1: Eine vollständige Methode nach Sturm und Shehory (2003)

Mangel an Methoden zur Entwicklung von MAS kritisiert:

> One of the most fundamental obstacles to the take–up of agent technology is the lack of mature software development methodologies for agent–based systems. Clearly, basic principles of software engineering ... need to be applied to the development and deployment of multi–agent systems, but they also need to be augmented to suit the differing demands of this new paradigm.

Die Autoren kommen zu dem Schluss, dass die Entwicklung von agentenbasierten Anwendungen meist ad hoc geschieht. Nur ansatzweise werden systematische Vorgehensweisen verwendet, was die Qualität der erstellten Anwendungen beeinträchtigt. Für Sie ist dies einer der Hauptgründe, warum die erstellten Systeme industriellen Anforderungen nicht genügen können. Außerdem bieten systematische Entwicklungsmethoden meist umfangreiche Unterstützung für Softwareprojekte insgesamt. Oft werden das Management eines Projektes und weitere Tätigkeiten, wie Testen, Wartung, etc., von ihnen angeleitet.

Wooldridge und Jennings (1998, 1999) warnen davor, agentenbasierte Entwicklungsansätze zu überschätzen. Sie identifizieren eine Reihe von Problemen, die eine agentenbasierte Softwareentwicklung behindern, bzw. die Qualität des entstehenden Systems senken, können. Diese klassifizieren sie in sechs Gruppen. Die *political pitfalls* treten auf, wenn dem Agenten–Paradigma zu viel zugetraut wird, es als alleinige Lösung von Problemen angesehen wird. Die sog. *management pitfalls* weisen darauf hin, dass sich ein Projekt genau im klaren sein sollte, warum es Agenentechnologie einsetzen will und warnen davor, eine zu generische Software entwickeln zu wollen. Die *conceptual pitfalls* weisen darauf hin, dass auch Agentensysteme systematisch entwickelt werden müssen. *Analysis and design pitfalls* treten auf, wenn verwandte Technologien ignoriert werden, die Nebenläufigkeit der Agenten nicht berücksichtigt wird oder die Einbettung von Legacy–Systemen nicht erwogen wird. Die *agent–level pitfalls* weisen darauf hin, dass der Einsatz von KI-Techniken in den Agenten gut überlegt werden muss, und es nicht immer nötig ist, eine eigene Agenten–Architektur zu entwickeln. Zuletzt beschäftigen sich die *society–level pitfalls* mit der Architektur einer Organisation von Agenten. Die Entscheidung, wie

viele Agenten das System bilden sollen, ist schwer zu treffen. Die Autonomie der Agenten soll nicht zu einem unstrukturierten System missverstanden werden, die Interaktionen sollen nicht zu kompliziert werden.

Diese häufig gemachten Fehler weisen darauf hin, dass ein genaues Verständnis von Nöten ist, was Agenten ausmacht, und wie sie eingesetzt werden können. Methoden leiten Entwickler in einer bestimmten Vorgehensweise an, wie dieses neue Paradigma zu verwenden ist. Die zielgerichtete und systematische Entwicklung, wie sie durch Methoden (an–)geleitet wird, soll helfen diese Probleme zu vermeiden.

3.3 Vorgeschlagene Methoden

Ausgehend von den eben beschriebenen Überlegungen, wurden eine Reihe von Methoden zur Entwicklung von Agentensystemen vorgeschlagen. Iglesias, Garijo und González (1999) unterscheiden die Ansätze zur agentenorientierten Entwicklung in vier Gruppen:

- Weiterentwicklungen objektorientierter Methoden
- Weiterentwicklungen von Methoden des *Knowledge Engineering*
- Weiterentwicklungen Formaler Beschreibungen
- Weitere Ansätze

Sie betonen, dass die Anstrengungen sich vor allem auf die Weiterentwicklung der ersten beiden Ansätze konzentrieren. Die Weiterentwicklungen objektorientierter Ansätze sind sehr zahlreich, auf die Unterschiede zwischen Agenten und Objekten wurde bereits in Abschnitt 2.2 eingegangen. Das *Knowledge Engineering* entstand in den 80'er Jahren als Teildisziplin der Künstlichen Intelligenz. Es beschäftigt sich mit der systematischen und kontrollierten Entwicklung von wissensbasierten Systemen (Studer et al., 2000 geben einen Über- und Ausblick). Hierbei stehen die Erfassung von Wissen, dessen Abbildung, Verarbeitung und Darstellung im Vordergrund. Ein Beispiel dieser Ansätze zur Systementwicklung ist die Methode *CommonKADS*[2] (eine kurze Darstellung ist bei Avison und Fitzgerald (2003) zu finden). Diese Methode verwendet einen eigenen Begriff der Agenten, im Sinne von Akteuren, die Teile eines Geschäftsprozessen ausführen. Diese Ansätze, in denen bereits eine Vorstellung von Agenten existierte, wurden dann zu agentenorientierten Methoden weiterentwickelt. Da diese Ansätze die oben beschriebene BDI–Architektur nicht explizit unterstützen, werden sie hier nicht weiter betrachtet.
Die Versuche, Agentensysteme formal zu spezifizieren, werden in dieser Arbeit ebenfalls nicht weiter untersucht. Sie könnten alternative Notationen bereitstellen, bzw. zur Verifizierung für kritische Applikationen benutzt werden, aber als Methoden, wie sie in Abschnitt 3.1 dargestellt wurden, sind sie nicht zu verstehen. Zusätzlich gib es eine Reihe von "Eigenentwicklungen", die sich aus speziellen Anwendungskontexten oder Erfahrungen mit konkreten Plattformen entwickelten.

Tabelle 3.1 zeigt eine List der gefundenen agentenorientierten Methoden.[3] Die Quellenangaben zeigen, dass es sich hauptsächlich um jüngere Vorschläge handelt.

[2] http://www.commonkads.uva.nl/frameset-commonkads.html

[3] eine umfangreiche, wenn auch nicht vollständige Liste von Methode ist unter: http://www.science.unitn.it/ recla/aose/ zu finden

Methode	Quelle(n)
AAII Methodology (AAII: Australian AI Institute)	Kinny, Georgeff und Rao, 1996
AALAADIN	Ferber und Gutknecht, 1998
Adept	Jennings et al., 2000
Agent Oriented Analysis and Design	Burmeister, 1996
Agent Oriented Methodology for Enterprise Modelling	Kendall, Malkoun und Jiang, 1996
Agent–SE	Far, 2001
Agent–UML (AUML)	Odell, Parunak, und Bauer, 2000
Atelier pour le Développement de Logiciels à Fonctionnalité Emergente (Adelphe)	Bernon et al., 2002
Cassiopeia	Collinot, Drogoul und Benhamou, 1996
Conceptual Modelling of Multi Agent Systems (CoMoMas)	Glaser, 1997
DEsign and Specification of Interacting REasoning components (DESIRE)	Jonker, Klush und Treur, 2000
Gaia	Wooldridge, Jennings und Kinny, 2000
Ingenias	Gomez-Sanz und Fuentes, 2002
MAS CommonKADS (CommonKADS: Common Knowledge Analysis and Design System	Iglesias et al., 1998
Methodology for Agent-Oriented Analysis and Design (AOM)	Wooldridge, Jennings und Kinny, 1999
Methodology for Engineering Systems of Software Agents (MESSAGE[/UML])	Caire et al., 2001a; 2001b
Multi Agent Scenario Based Method (MASB)	Moulin und Brassard, 1996
Multiagent Systems Engineering Methodology (MaSE)	DeLoach, 2001; Wood und DeLoach, 2000
Multi Agent SystemS Iterative View Engineering (MASSIVE)	Lind, 2001
Open Distributed Applications Construction) (ODAC)	Gervais, 2002
OPEN agents (OPEN: Object-oriented Process, Environment and Notation)	Debenham und Henderson-Sellers, 2002
OPM/MAS (OPM: Object-Process Methodology)	Sturm, Dori und Shehory , 2003
Process for Agent Societies Specification and Implementation (PASSI)	Cossentino und Potts, 2002
Prometheus	Padgham, 2002; Padgham und Winikoff, 2002
Role Oriented Analysis and Design for Multi-Agent Programming (ROADMAP)	Juan, Pearce und Sterling, 2002
Standards Based and Pattern Oriented Multi-Agent Development Methodology (SABPO)	Dikenelli und Erdur, 2002
Societies in Open and Distributed Agent spaces (SODA)	Omicini, 2000
STYX Agent Methodology	Bush, Cranefeld und Purvis, 2001
Tropos	Giunchiglia, Mylopoulos und Perini, 2001
Zeus Methodology	Nwana et al., 1999

Tabelle 3.1: Liste agentenorientierter Methoden

In ihrem Resumé über eine Diskussion in einem Workshop der OOPSLA 2002 stellten Henderson-Sellers und Gorton (2003) die Entwicklung verschiedener Erweiterungen objektorientierter Methoden, in einer Genealogie dar. Dort wurden die, in der Konferenz erörterten Methoden eingezeichnet. Sie unterscheiden zwischen den Erweiterungen objektorientierter Methoden und unabhängigen Weiterentwicklungen. Abbildung 3.2 stellt in entsprechender Weise die in Tabelle 3.1 genannten Methoden in Beziehung zueinander.
Die Ovale bezeichnen Konzepte bzw. Methoden, die um agentenorientierte Konzepte erweitert wurden. Deutlich ist der überwiegende Anteil von Weiterentwicklungen objektorientierter Methoden (als *OO* oben links bezeichnet) zu erkennen. Die gestrichelten Kästen markieren konkrete, nicht agentenorientierte Methoden, die als Ausgangspunkte für Weiterentwicklungen dienten. Eine Reihe von Eigenent-

wicklungen verstehen sich als direkte Weiterentwicklungen objektorientierter Vorgehensweisen und Erfahrungen, ohne von solchen konkreten Methoden beeinflusst worden zu sein. Die kleineren gestrichelten Kästen bezeichnen einzelne Techniken, die angepasst wurden. Als weitere Ausgangspunkte sind *KE* für das Knowledge En-

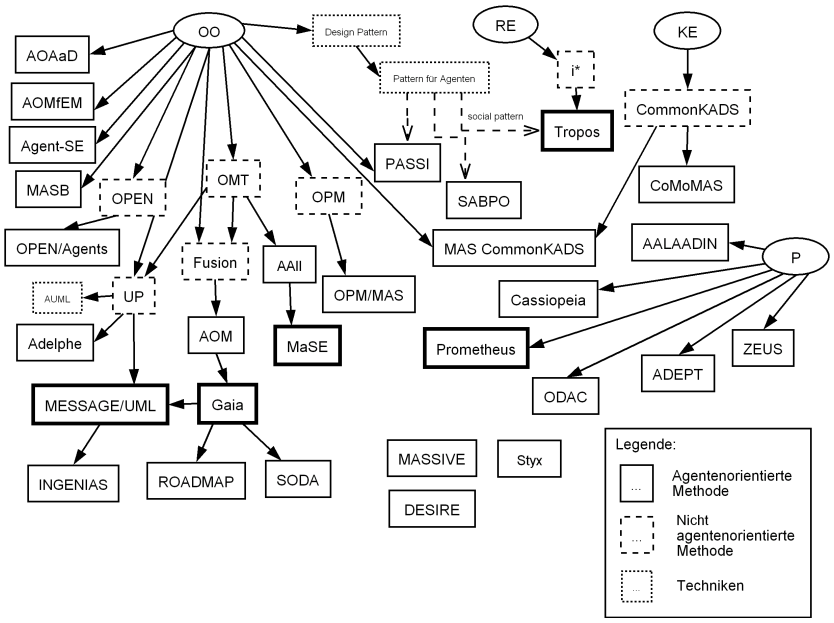


Abbildung 3.2: Genealogie der verschiedenen agentenorientierten Methoden

gineering und *RE* für *Requirements Engineering* aufgetragen. Die oben erwähnte Methode CommonKADS unterstützt dieses Knowledge Engineering. Zwei Erweiterungen dieses Ansatzes wurden vorgeschlagen. Die konkrete Vorgehensweise *i** konzentriert sich auf die Erhebung der sog. "frühen" Anforderungen an ein Softwaresystem. Sie hat die Tropos–Methode stark beeinflusst und wurde deshalb in die Übersicht aufgenommen (Das Framework i* und diese "frühen" Anforderungen werden in Abschnitt 5.3.4 vorgestellt). Eine weitere Gruppe sind die Entwicklungen von Methoden, die sich aus Erfahrungen mit einer konkreten Plattform oder Referenz–Architektur (als *P* aufgetragen; rechte Seite) entwickelt haben. In der unteren rechten Ecke sind verbleibende Eigenentwicklungen eingezeichnet, die sich nicht direkt von anderen Ansätzen der Modellierung haben inspirieren lassen. Die fett umrahmten Methoden werden im Verlauf dieser Arbeit noch eingehender betrachtet werden. Sie sind am weitesten verbreitet, gut dokumentiert und unterstützen die BDI–Architektur.

Kapitel 4

Auswahl einer Methode

Ziel des Vergleiches von Methoden ist (1) die Entwicklung eines besseren Verständnisses der Eigenschaften und Unterschiede der einzelnen Methoden und (2) die Untersuchung ob die Methoden für das Jadex–System benutzbar sind. Es wird davon ausgegangen, dass nur rein kommunikative Agentensysteme entwickelt werden sollen (siehe Abschnitt 2.4). Vor allem in den 90'er Jahren entstanden zahlreiche Arbeiten, die versuchten, Richtlinien für die Bewertung von objektorientierten Methoden zu finden. Diese Ansätze werden kurz charakterisiert, um in die Thematik des Vergleiches von Methoden einzuführen. Außerdem stellen diese Überlegungen Ausgangspunkte für (eine überschaubare Anzahl von) Arbeiten dar, die bereits versuchten, agentenorientierte Methoden miteinander zu vergleichen. Auch diese Arbeiten werden kurz vorgestellt, und ihre Ergebnisse charakterisiert. Diesen Arbeiten wurden Kriterien entnommen, um die Eignung für ein spezielles System zu untersuchen. Sie werden im letzten Teil dieses Kapitels vorgestellt.

4.1 Vergleich und Bewertung von Methoden

Im Laufe der 90'er Jahre wurden eine Reihe von Softwareentwicklungsmethoden für objektorientierte Systeme entwickelt. In diesem Zusammenhang entstanden zahlreiche Arbeiten, die Richtlinien und Vorgehensweisen entwickeln, um diese Methoden miteinander zu vergleichen.
Am Software Engineering Institute[1] (SEI) der Carnegie Mellon University haben Wood et al. (1988) einen fünfstufigen Prozess zur Beurteilung von Methoden für die Entwicklung von Echtzeitsystemen entwickelt.
Sie geben fünf grundlegende Gesichtspunke vor, unter denen Methoden zu betrachten sind. Für diese werden Fragenkataloge angegeben, die bei einer Bewertung zu beantworten sind. In dieser frühen Arbeit wird unter der Entwicklung eines Systems lediglich die Spezifikation und das Design verstanden. Die Durchführung von Fallstudien wird als Grundlage zur Beantwortung der Fragen vorgeschlagen.
In der ersten Phase werden die wichtigen Charakteristika des zu entwickelnden Systems bestimmt und betrachtet, wie diese in den einzelnen Methoden berücksichtigt werden. Anschließend werden Beschränkungen identifiziert, die dem zu entwickelnden System auferlegt sind und wiederum ermittelt, wie sich diese auf die Benutzung

[1] http://www.sei.cmu.edu/

der Methoden auswirken. In der dritten Phase werden dann Eigenschaften und Benutzbarkeit der Methoden untersucht. Es wird betrachtet, wie Systeme dargestellt werden und welche Richtlinien für Ableitung und Untersuchung der einzelnen Modelle, vorgegeben werden. Die vierte Phase untersucht, inwieweit das Management eines Softwareprojektes unterstützt wird. Dies schließt eine Abschätzung von Kosten und Nutzen des Einsatzes mit ein. Zuletzt werden die Unzulänglichkeiten und mögliche Anpassungen der Methode identifiziert.

Berard (1995) vergleicht mehrere objektorientierte Methoden miteinander, um Firmen in der Integration der aufkommenden objektorientierten Methoden in ihre Unternehmensstrukturen zu unterstützen. Bewertungen von Methoden nach den von ihm identifizierten Gesichtspunkten sollen dazu dienen, diese neuen Technologien zu verstehen, eine besser geeignete Methode zum Einsatz in einem bestimmten Unternehmen zu finden, und die Ablehnung des Einsatzes von Methoden für ein Unternehmen zu begründen.
Er stellt 6 Gesichtspunkte vor, die eingehend analysiert werden müssen, um Methoden voneinander abgrenzen zu können. *Concepts* beschreiben, welche Konzepte der (aufkommenden) Objektorientierung von den Methoden unterstützt werden. Die Notationen (*Notations*) sollen daraufhin untersucht werden, welche verschiedenen Aspekte eines Systems modelliert werden, und wie sie den Entwicklungsprozess leiten. Das Vorgehensmodell (*Process*) wird in Bezug auf seine Vollständigkeit, seine Beschaffenheit und seine Unterstützung des Projektmanagements untersucht. Unter (*Pragmatics*) wird die vorhandene Dokumentation für die Methoden, ihre etwaige Spezialisierung auf gewisse Anwendungsgebiete und die vorhandene Werkzeugunterstützung betrachtet. *Support for Software Engineering Principles and Goals* fasst Kriterien des Software–Engineering zusammen, die unabhängig von objektorientierten Konzepten sind (Wiederverwendung, Modularisierung, etc.). Zuletzt wird unter *Marketability* untersucht, wie die Methoden in eine Organisation eingeführt und angenommen werden können. Außerdem identifizierte Berard Schwierigkeiten, die beim Vergleich von Methoden auftreten. Zum einen ergeben sie sich aus den großen Unterschieden zwischen einzelnen Methoden und Terminologien, zum anderen können sich "Formfehler" einschleichen, die die Ergebnisse verändern.

Desmet ist der Name einer Methode die von Kitchenham (1996) entwickelt wurde, um Vergleiche durchzuführen. Sie gibt Richtlinien zur Auswahl einer geeigneten Vorgehensweise zur Evaluierung und Durchführung von qualitativen und quantitativen Analysen vor. Es werden drei grundlegende Untersuchungsarten beschrieben und ihre Durchführung angeleitet. Quantitative Untersuchungen können mittels eines formalen Experimentes, einer Fallstudie und einer Umfrage durchgeführt werden. Hierbei werden messbare Auswirkungen der Benutzung einer Methode betrachtet. Qualitative Untersuchungen (genannt *Feature Analysis*) basieren auf der Identifizierung von Anforderungen, die Anwender an die zu benutzende Methode stellen. Anschließend werden die Methoden daraufhin untersucht, inwieweit sie diese Anforderungen erfüllen. Diese Untersuchungen können im Zuge eines Screenings, eines Experimentes, einer Fallstudie und einer Umfrage durchgeführt werden. Screening bezeichnet hierbei die Untersuchung durch eine einzelne Person. In einem Experiment führte eine Gruppe von potentiellen Benutzern die Untersuchung unter kontrollierten Bedingungen durch. Als weitere Möglichkeiten werden zwei sog. Hybride Methoden genannt. In der *qualitative effects analysis* wird eine Expertenmeinung über die qualitativen Auswirkungen des Einsatzes einer Methode eingeholt. Ein *Benchmarking*

kann üblicherweise nur eingesetzt werden, um Werkzeuge zu betrachten.

Fazit

Zur Gegenüberstellung der Methoden wurde ein Screening durchgeführt. Der zeitliche Rahmen und die Mittel, die für diese Diplomarbeit zur Verfügung standen bedingen diesen Ansatz. Wie später dargestellt wird, sind einige Aspekte der betrachteten Methoden nicht objektiv durch eine Analyse zu erfassen. Zu deren Bewertung werden Fallbeispiele entwickelt. Der nächste Abschnitt betrachtet bereits durchgeführten Untersuchungen von agentenorientierten Methoden, um einen geeigneten Kriterienkatalog zusammenzustellen.

4.2 Vergleiche agentenorientierter Methoden

Es wurde bisher eine überschaubare Anzahl an Arbeiten zum Vergleich agentenorientierter Methoden veröffentlicht (Shehory und Sturm, 2001; O'Malley und DeLoach, 2001; Cernuzzi und Rossi, 2002; Dam und Winikoff, 2003; Sturm und Shehory, 2003). Diese vergleichen jeweils eine kleine Anzahl von Methoden, aufgrund eigens aufgestellter Kriterienkataloge. Zwei Arten von Kriterien werden in allen Fällen betrachtet. Zum einen werden die Kriterien der Bewertung von objektorientierten Methoden berücksichtigt, zum anderen die Unterstützung agentenorientierter Konzepte. Die verwendeten Kriterienkataloge, sind unabhängig von Anwendungsfeldern und Zielplattformen.

O'Malley und DeLoach, (2001), beschäftigten sich indirekt mit dem Vergleich agentenorientierter Methoden. Sie entwickelten eine Menge von Kriterien, mit deren Hilfe entschieden werden soll, ob eine objektorientierte oder ein agentenorentierte Methode für ein konkretes Softwareprojekt am geeignetsten ist. Sie beziehen sich dabei auf Arbeiten vom SEI der Carnegie–Mellon University (Wood et al., 1988). Die dort genannten Kategorien der Kriterien werden von ihnen in *Management Issues* und *Project Requirements* unterteilt.
Die Management Issues betrachten hauptsächlich die Auswirkungen der Wahl einer Methode für eine (Software erstellende) Organisation. Gesichtspunkte, wie die Kosten einer Umstellung auf eine neue Methode und deren Benutzbarkeit innerhalb der Organisation werden betont. Die Project Requirements fassen die technischen Eigenschaften einer Methode zusammen. Die Autoren fanden eine Menge von Eigenschaften eines Softwaresystems, die modellierbar sein sollten (z. B. Interaktionen, Verteilung, etc.).
Diese Kriterien wurden dann in einer elektronischen Umfrage (zwischen November und Dezember 2000; über Mailing–Listen der OMG, Newsgroups comp.ai und comp.software-eng, etc.) zur Diskussion gestellt. Aufgrund der Rückmeldungen wurden die Kriterien der beiden Kategorien überarbeitet und deren Gewichtung festgelegt. Diese Kriterien werden mittels der *Multiobjective Decision Analysis* (beschrieben von Kirkwood 1997; nach O'Malley und DeLoach, 2001) zu einer objektiven Bewertungsnote geführt. Die einzelnen Kriterien werden nach Noten bewertet, das gewichtete Mittel dieser Werte zu berechnet, und die so gewonnenen Kennzahlen der einzelnen Methoden einander gegenüber gestellt. Als Anwendungsbeispiel stellen die Autoren die objektorientierte Booch–Methode (Booch, 1994) der MaSE-Methode gegenüber (DeLoach ist einer der Entwickler der MaSE–Methode).

Die Wahl der Kriterien wurde mit großer Sorgfalt vorgenommen, sie sind aber zu allgemeimen gehalten, um in dieser Arbeit verwendet werden zu können. Da hier die Implementation mittels eines speziellen Agentensystems unterstützt werden soll sind die technischen Gesichtspunkte weit intensiver zu betrachten, als es in dieser Arbeit geschah. Kitchenham (1996:21) beschreibt die Ungenauigkeiten und Unzulänglichkeiten, die die alleinige Bewertung mittels Kennzahlen mit sich bringt.

Shehory und Sturm (2001) stellen eine Kriteriensammlung vor, die eine umfassende Bewertung von agentenorientierten Methoden ermöglichen soll. Aufgrund der genau identifizierten Unterschiede der einzelnen Methoden, stellen sie Schlussfolgerungen an, inwieweit die Methoden den Bedürfnissen von Entwicklern gerecht werden. Sie hoffen so Impulse zur Weiterentwicklung der Methoden geben zu können. Die gefundenen Kriterien werden unterschieden zwischen den *Software–Engineering Criteria* und den *Agent–Based Characteristics*.
Drei Methoden (Gaia, Adept und Desire; zu finden in Tabelle 3.1 und Übersicht 3.2) werden bezüglich dieser Kriterien bewertet. Die Erfüllung der einzelnen Kriterien wird prosaisch wiedergegeben, eine Tabelle listet die Unterstützung der Eigenschaften auf (in einer Klassifizierung von gut, zufrieden stellend, nicht zufrieden stellend, nicht unterstützt).
Nach der Bewertung kommen die Autoren zu dem Schluss, dass mehrere Eigenschaften von herkömmlichen Methoden des Software–Engineering (Ausführbarkeit/Testen, Verfeinerung, Analysierbarkeit der Modelle) kaum unterstützt werden. Auch die Verteilung und die Beschreibung der Kommunikation zwischen Agenten soll ausdrucksstärker modellierbar sein. Diese Zusammenfassung der Ergebnisse gibt einen Eindruck von der Art der in Betracht gezogenen Kriterien.

Die angewendeten Kriterien waren zu allgemein gefasst, um eine detaillierte Bewertung hinsichtlich einer speziellen Plattform zu ermöglichen, gaben aber einen Überblick darüber, was betrachtet werden sollte, um die Unterschiede zwischen Methoden zu verstehen.

Auch Cernuzzi und Rossi (2002) schlagen eine qualitative Analyse mit anschließender quantitativer Bewertung vor. In dem von ihnen vorgeschlagenen Bewertungsprozess wird, nachdem die Ziele der Bewertung definiert wurden, ein sog. *Attributes Tree* aufgestellt. Die Blätter dieses Baumes stellen die zu quantifizierenden Kriterien dar. Gewichte an den Zweigen geben an, wieviel die einzelnen Kriterien zu ihren Elternknoten beitragen. Diese Gewichtung zieht sich durch den gesamten Baum, bis letztendlich der Wert der Wurzel berechnet werden kann. Nach einer Bewertung der einzelnen Kriterien werden die Werte der Wurzeln einander gegenüber gestellt. In ihrer Arbeit wird nur die Modellierung von Agentensystemen untersucht. Die Autoren identifizieren drei Arten von Kriterien, die hierzu in Betracht gezogen werden. Die *Internal Attributes* beschäftigen sich mit dem internen Aufbau der Agenten, *Interaction Attributes* beschreiben wie die Interaktionen zwischen Agenten modelliert werden können und die *Other Process Requirements* bewerten die Design– und Entwicklungprozesse, die die Methoden vorschlagen. Als Anwenundgsbeispiel stellen sie die frühe AAII–Methode dem MAS–CommonKADS gegenüber (beide ebenfalls zu finden in Tablelle 3.1 und Übersicht 3.2).

Die Autoren betonen die allgemeine Anwendbarkeit ihres Ansatzes zur Bewertung

von Methoden. Nachdem die Ziele einer Bewertung klar definiert sind, sind Anwender dieses Schema frei, ihre eigenen Attribut–Bäume auszustellen. Da die Bewertung letztendlich wieder auf einzelne Zahlenwerte zurückgeführt wird, muss sich dieser Ansatz ebenfalls der Kritik an solchen Vorgehensweisen von Kitchenham (1996:21) stellen. Allerdings sind die gewonnenen Zahlenwerte hier (durch die Baumstruktur der Gewichte) leichter nachvollziehbar. Die Subjektivität der Bewertung wird von den Autoren nicht betrachtet.

Die eben vorgestellten Arbeiten haben die Eigenschaften der Methoden durch Literaturrecherche identifiziert und bewertet. Von Dam und Winikoff (2003) (basierend auf einer Arbeit von Dam, 2003) wurde die detaillierteste Liste von Kriterien erstellt. Eine Anzahl von Methoden (Gaia, MESSAGE, MaSE, Prometheus und Tropos; siehe Tablelle 3.1 und Übersicht 3.2) wurde ausgewählt, um sie möglichst umfassend zu bewerten. Für die Auswahl dieser fünf Methoden waren, die gefundene Dokumentation, ihre allg. Anerkennung als agentenorientierte Methoden und ihre Weiterentwicklung, basierend auf Erfahrungen von Benutzern, ausschlaggebend. Die in Betracht gezogenen Kriterien werden in vier Gruppen unterteilt (diese Einteilung basiert auf dem Bewertungsschema das Berard (1995) aufgestellt hat):

Concepts (Konzepte) zur Bewertung der Unterstützung agentenorientierter Konzepte.

Modelling Language (Modellierungssprache) zur Bewertung der Benutzbarkeit und Ausdruckskraft der verwendeten Notationen.

Processes (meint hier: Verfahren zur Entwicklung) zur Bewertung des Vorgehensmodelles und der Beschreibung der einzelnen Verfahren, die in der Entwicklung verwendet werden sollen

Pragmatics (Pragmatische Gesichtspunkte) zur Bewertung der Unterstützung des Projektmanagements und weiterer technischer Eigenschaften des Systems (Konzentration auf gewisse Anwendungsbereiche, Verteilung etc.).

Mehrere Studenten wurden damit beauftragt ein bestimmtes System, mit den jeweiligen Methoden zu modellieren (Dezember 2002 bis Februar 2003). Ihre Erfahrungen gaben sie in Fragebögen wieder. Auch die Autoren der einzelnen Methoden wurden so befragt.

Der Kriterienkatalog war sehr ausführlich, und gab einige Anregungen für die in dieser Arbeit angewandten Kriterien. Die Untersuchungen der Concepts und Processes gleichen den anderen oben aufgeführten Arbeiten dahingehend, dass es sich um objektiv zu bewertende Eigenschaften handelt. Die Unterstützung einzelner agentenorientierter Konzepte kann leicht aus der Dokumentation ersehen werden. Zur Bewertung der angewendeten Verfahren wurden diese dem Vorgehensmodell der Prometheus–Methode gegenübergestellt. Es wurde verglichen, ob die in Prometheus ausgeführten Arbeitsschritte (sowie zusätzlich Testen, Deployment und Wartung) von den Methoden unterstützt werden.
Es ist leicht einzusehen, dass eine Umfrage die einfachste Möglichkeit für die Bewertung der Modellierungssprache und der pragmatischen Gesichtspunkte war. Neben den, in den vorigen Arbeiten angewendeten Kriterien sollten die Modellierunssprachen auch dahingehend untersucht werden, wie verständlich und benutzbar sie sind. Dies würde zu ergonomischen Überlegungen führen, ebenso wie unter den

praktischen Gesichtspunkten die Benutzbarkeit der, soweit vorhandenen, CASE–Werkzeuge betrachtet werden sollte. Unter den praktischen Gesichtspunkten wurde auch die Unterstützung des Managements eines (agentenorientierten) Softwareprojektes untersucht, und die bereits stattgefundene Benutzung der Methoden in verschieden Projekten erhoben.
Diese Befragung von Benutzern der Methoden ist die erste Untersuchung, die so in einem größeren Rahmen stattfand. Besonders hilfreich war die ausführliche Erhebung der objektiven Eigenschaften und quantitativen Werte der Methoden. In der weiteren Bewertung der Methoden wird gelegentlich auf die dort erfassten Ergebnisse zurückgegriffen, da im Rahmen dieser Arbeit eine Befragung von Benutzern nicht möglich war. Dies hätte in einem ähnlich kontrollierten Rahmen mit mehreren Probanden, die das selbe Problem zu lösen versuchen, geschehen müssen.
Allerdings sind die Ergebnisse der Umfrage mit Bedacht zu bewerten. Die Methoden MaSE, Tropos und Prometheus werden von den Befragten als geeignet und anschaulich bewertet. Dies mag an ihren Eigenschaften liegen, oder an der Vorbildung der Befragten. Es handelte sich um Studenten der Universität, an dem Entwickler der Prometheus–Methode unterrichten. Die geringe Anzahl der befragten Studenten scheint keine weiteren Schlüsse zuzulassen.

In der aktuellsten Arbeit zu diesem Thema greifen Sturm und Shehory (2003) das Bewertungsschema aus der eben genannten Arbeit von Dam und Winikoff (2003) auf und vereinen es mit ihrer früheren Arbeit. Ihren überarbeiteten Kriterienkatalog unterteilen sie also in entsprechende Kategorien: *Concepts and Properties*, *Notations and Modeling Techniques*, *Process* und *Pragmatics*.
Dieses vorgeschlagene Bewertungsschema wurde im Hinblick auf drei Ziele entwickelt. Es soll (vornehmlich industriellen) Organisationen helfen, eine Methode für die Entwicklung eines Agentensystems auszusuchen, Entwicklern eigener Methoden benötigte Eigenschaften aufzeigen, und anhand dieser Eigenschaften, eine Auswahl an am weitesten entwickelten Methoden ermöglichen, um eine (mögliche) Standardisierung einzuleiten. An den früheren Arbeiten kritisieren sie hauptsächlich, dass nicht der gesamte Umfang an Eigenschaften, die eine Methode des Software–Engineering ausmachen, untersucht wurden. Eine ideale Methode nach ihrer Definition beinhaltet: einen Vorgehensmodell für den gesamten Lebenszyklus einer Entwicklung (full lifecycle process), eine umfassende Menge von Konzepten und Modellen, eine Menge von sog. Techniken zur Unterstützung der Entwicklung (hier verstanden als Regeln, Richtlinien und Heuristiken), eine abgegrenzte Menge auslieferbarer Artefakte, eine Modellierungssprache (bestehend aus Metamodel und Notation(en)), Metriken, Maßnahmen zur Qualitätssicherung, Standards (z.B. zur Codierung) und Projektmanagement. Abbildung 3.1 zeigt die Abhängigkeiten dieser Elemente untereinander.
Die Gaia Methode wird beispielhaft nach dem vorgestellten Kriterienkatalog bewertet. Die Ergebnisse zeigen, das die Methode insbesondere die neu betrachteten Eigenschaften von umfassenden Methoen nicht unterstützt.

Die Autoren zeigen mittels ihres Vergleiches Schwachstellen existierender Methoden auf und geben somit Anregungen für Weiterentwicklungen. Diese neue Idee, eines Ausblicks, auf das, was von Methoden zu erwarten sein sollte, ist wertvoll, trägt aber zu der Untersuchung, wie die Entwicklung eines Systems momentan unterstützt werden kann, wenig bei.

4.3 Die Bewertungskriterien

Das eben beschriebene Schema zur Einteilung, der zu bewertenden bzw. der von einer Methode gewünschten, Eigenschaften wird übernommen. Da in dieser Arbeit die Anwendbarkeit der Methoden hinsichtlich einer konkreten Plattform untersucht wird, konzentriert sich der folgende Kriterienkatalog auf die tatsächlich unterstützen Eigenschaften. Insbesondere aufgrund der gewonnenen Erfahrungen, bei der Benutzung der letztendlich ausgewählten Methode, werden im späteren Abschnitt 7.3.4 Unzulänglichkeiten gezeigt, die die deren Benutzung für das Jadex System erschweren, und somit auf wünschenswerte Erweiterungen hinweisen.

Kriterien der agentenorientierten Konzepte

Unter den agentenorientierten Konzepten werden diejenigen berücksichtigt, die für das Jadex–Systems erheblich sind. Neben den Konzepten, die zur Modellierung der einzelnen Agenten benutzt werden, werden auch die Mittel betrachtet, die die Kommunikation zwischen den Agenten beschreiben.

Die Beschreibungen der Agenten sollten folgende Konzepte unterstützen:

BDI–Architektur (hier: Beliefs, Goals, Plans) Die Methoden müssen die BDI–Architektur unterstützen. Das einzelne Wissen eines Agenten soll repräsentiert werden, die Ziele und Pläne eines Agenten müssen modelliert werden. Es sollte beschrieben werden, wie (meint durch welche Pläne) die Ziele erfüllt werden, und welche Pläne auf welche Beliefs zugreifen.

Capabilities Dieses Konzept zur Modularisierung von Agenten sollte unterstützt werden. Die Sichtbarkeit der enthaltenen Beliefs, Goals und Pläne sollte leicht ersehbar sein.

Autonomie Die Fähigkeiten eines Agenten, autonom Probleme zu lösen wird unterstützt durch Beschreibungen von Aufgaben oder Tätigkeiten (functionalities oder tasks), die ein Agent eigenverantwortlich ausführen kann. Außerdem sollten die Mechanismen, die im Agenten benutzt werden, um Entscheidungen zu treffen, ausdrückbar sein.

Proaktivität Es muss ausdrückbar sein, welche Ziele ein Agent, wie erfüllen kann. Die Modellierung der einzelnen Ziele ist ein erster Schritt hierzu. Wichtig ist, wie die Pläne eines Agenten in Beziehung zu den Zielen gebracht werden.

Events Dies unterstützt die Beschreibung der Reaktivität eines Agenten. Veränderungen in der Umgebung eines Agenten sollten durch Events modellierbar sein. Auch die Veränderungen innerhalb eines Agenten müssen durch Events ausdrückbar sein. Die Art eines Events sollte so genau beschreibbar sein, dass auf die Filter geschlossen werden kann, die Jadex benutzt, um Events zu unterscheiden.

Rollen Rollen beschreiben abstrakt die Funktion(en) eines Agenten. Ein Agent kann mehrere Rollen implementieren, und verhält sich in einem Kontext entsprechend einer dieser Rollen. Dies ist auch eine Art Strukturierungskonzept (ähnlich den Interfaces objektorientierter Sprachen). Eine Unterstützung ist wünschenswert.

Verteilung Da in den Multiagentensystemen die einzelnen Agenten verteilt sind und mobil sein können, ist eine geeignete Darstellung der Verteilung der einzelnen Agentenplattformen und Agenten (sowie deren Mobilität) nötig.

Protokolle Zur Beschreibung der Kommunikation innerhalb des Agentensystems, sollten diese zwischen den Agenten genau spezifizierbar sein. Dieses allgemeine Konzept bedarf einer agentenorientierten Semantik. Die Beschreibung der Protokolle ist eng verbunden mit den ausgetauschten Nachrichten.

Nachrichten Agenten kommunizieren miteinander über Nachrichten. Diese Nachrichten sollten als ACL–Nachrichten beschrieben werden, da diese von Jadex benutzt werden.

Kriterien für die Modellierungssprache

Die verwendete Notation stellt die abstrakte Sicht auf die wichtigen Aspekte des zu entwickelnden Systems dar. Sie besteht aus Symbolen, Syntax und Semantik. Die **Benutzbarkeit** (und auch Erlernbarkeit) wird durch eine eindeutige und verständliche Notation erhöht. Die Modelle sollten leicht zu zeichnen (möglichst auch auf Tafel, Papier) und möglichst intuitiv verständlich sein. Um sowohl die Erhebung der Anforderungen, die Analyse und das Design eines Softwaresystems zu unterstützen, sollte eine **ausdrucksstarke** Modellierungssprache verschiedene Sichten auf das System erlauben. Die funktionalen, strukturellen (Aggregationen, Spezialisierungen, etc.) und dynamischen (Interaktionen, zeitliches Verhalten, Zustandsänderungen, etc.) Eigenschaften sollten darstellbar sein. Die im System enthaltenen Daten (und auch der Datenfluß im System) gehören ebenfalls zur Struktur des Systems. Außerdem sollten die Modelle gewisse technische Eigenschaften unterstützen. Im laufe der Entwicklung sollte die **Verfeinerung** und **Modularisierung** der einzelnen Modelle unterstützt werden. Die Modelle sollten aufeinander **aufbauen** und einzelne Artefakte sollten **durchgängig** (traceable) sein, um sie in den verschiedene Modellen verfolgen zu können. Es ist unabdingbar, dass Syntax und Semantik **eindeutig definiert** sind.

Kriterien für die Vorgehensweise

Methoden schlagen neben einer geeigneten Notation auch eine Vorgehensweise zur Entwicklung eines Softwaresystems vor. Um die verschiedenen **Verfahren zur Entwicklung** miteinander zu vergleichen, werden sie in dieser Arbeit den Arbeitsschritten des "Unified Process" (UP) (Jacobson, Booch und Rumbaugh, 1999) gegenübergestellt. Die Fragestellung ist, welche Arbeitsschritte (Workflows) des RUP in dem jeweils vorgeschlagenen Modell **abgedeckt** werden. Die Gegenüberstellung zum UP ist willkürlich gewählt, die Autoren der oben genannten Arbeiten verwenden meist andere Modelle (vornehmlich Phasenmodelle aus ihren eigenen Entwicklungen). Ein solcher Vergleich mit den Phasen des UP wurde ebenfalls von Sturm und Sheory (2003) vorgeschlagen. Im Verlauf der vorliegenden Arbeit geht es lediglich darum, eine (dem Leser) möglichst bekannte Referenz zu verwenden, an der sich ein Vergleich orientieren kann.
Mit bedacht wurde der UP als Referenz gewählt, da die in Betracht kommenden Methoden diesem einfachen Prozess am nächsten kommen. Er wurde weiterentwickelt zum *Rational Unified Process* (RUP; beschrieben im Rational Software White

Paper 2001a), der sich durch eine größere Anzahl von Arbeitsschritten auszeichnet. Insbesondere das Management eines Softwareprojektes wird hier unterstützt. Da aber alle zu betrachtenden Methoden weit davon entfernt sind, eine derartige Unterstützung zu liefern, ist ein Vergleich mit dem 'kleinen Bruder' UP angebracht. Nach dem UP werden in vier Phasen die verschiedenen Kernarbeitsschritte (*Anforderungen*, *Analyse*, *Design*, *Implementierung* und *Testen*) ausgeführt.

Anforderungen Die Anforderungen an die Software werden ermittelt und Dokumentiert.

Analyse Der Problembereich wird weiter untersucht, um die Anforderungen an das System besser zu verstehen. Es werden keine Entscheidungen bezüglich der Implementation getroffen.

Design Es wird festgelegt, wie die Software zu implementieren ist.

Implementierung Umsetzung des Designs.

Testen Entwicklung von Testfällen (evtl. Testkomponenten, Testtreibern, etc.), Durchführung der Tests und Debugging.

Weiterhin von Interesse sind die **Eigenschaften** des vorgeschlagenen Vorgehensmodells (ob es iterativ ist oder nicht; Top–Down oder Bottom–Up).

Außerdem ist zu berücksichtigen, inwieweit ein **Management** des Softwareprojektes unterstützt wird. So könnten Heuristiken oder Richtlinien, beispielsweise bei der Aufwandsabschätzung oder der Qualitätssicherung hilfreich sein. Die **Komplexität** des Entwicklungsprozesses ist die letzte Eigenschaft, die in dieser Kategorie berücksichtigt werden soll. Die Aufgaben der einzelnen Arbeitsschritte und der Sinn ihrer Abfolge aufeinander sollten verständlich sein.

Kriterien für die Pragmatischen Gesichtspunkte

In erster Linie wird an dieser Stelle die verfügbare **Werkzeugunterstützung** charakterisiert. Wünschenswert sind Werkzeuge, die nicht nur leicht benutzbar sind, und den gesamten Entwicklungsprozess unterstützen, sondern auch die Konsistenz der entwickelten Modelle (soweit möglich) überprüfen. Die Benutzbarkeit eins Werkzeugs hängt zum großen Teil von seinen ergonomischen Eigenschaften ab. Die DIN–Norm 9241-10 (DIN, 1996) definiert sieben Eigenschaften, die die ergonomische Benutzbarkeit eines Werkzeugs beeinflussen. Diese sind Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit, Steuerbarkeit, Erwartungskonformität, Fehlertoleranz, Individualisierbarkeit und Lernförderlichkeit. Prümper und Anft (im Weiteren ISONORM–Fragebogen genannt, 1993) entwickelten einen (Multiple–Choice–) Fragebogen (in die Benutzung führt Bräutigam (1999a, 1999b) ein), um die Erfüllung dieser Eigenschaften zu überprüfen. Die einzelnen Fragen konzentrieren sich auf die Bewertung von umfangreichen Benutzungsschnittstellen. Daher sind einige der Fragen unpassend für die im Folgenden betrachteten Modellierungswerkzeuge. Die Fragen und vor allem die betrachteten Eigenschaften geben aber einen Eindruck davon, unter welchen Gesichtspunkten die Werkzeuge zu betrachten sind (Für eine vollständige Auflistung der einzelnen Fragen sei der Leser auf Anhang D verwiesen). Die **Dokumentation** zu den einzelnen Methoden ist ein weiterer wichtiger Faktor

zu ihrer Anwendbarkeit. Die Vorgehensmodelle und einzelnen Arbeitsschritte werden üblicher Weise durch Heuristiken und Beispiele beschrieben. Zum Verständnis zukünftiger Benutzer trägt eine verständliche Dokumentation der einzelnen Schritte erheblich bei. Auch die Dokumentation der Werkzeuge (wenn vorhanden), Notationen, etc. sollte so verständlich wie möglich sein. Bekannte Erfahrungen mit der **Benutzung** der Methoden in Projekten sind auch zu berücksichtigen. Alle hier bewerteten Methoden sind Neuentwicklungen, daher sind die bereits gesammelten Erfahrungen mit deren Anwendung relevant. Eine Methode die bereits in größeren Projekten eingesetzt wurde, ist einer unerprobten Methode vorzuziehen.

Tabelle 4.1 listet die beschriebenen Kriterien noch einmal auf:

Konzepte	Modellierungssprache	Vorgehensweise	Pragmatische Gesichtspunkte
BDI–Architektur Capabilities Autonomie Proaktivität Events Rollen Verteilung Protokolle Nachrichten	Benutzbarkeit Ausdrucksstärke Verfeinerung Aufbau der Modelle Durchgängigkeit Eindeutige Definition Modularisierung	Abdeckung der Arbeitsschritte Management Komplexität Eigenschaften des Vorgehens	Werkzeugunterstützung Dokumentation Benutzung in Projekten

Tabelle 4.1: Übersicht der Bewertungskriterien

4.4 Vorgehensweise bei der Bewertung

Aufgrund der gefundenen Kriterien soll nun eine Bewertung der gefundenen Methoden vorgenommen werden. Die Unterstützung der Konzepte und Vorgehensweise, kann objektiv (nach einem Screening) festgestellt werden. Wohingegen die Modellierungsprache und die pragmatischen Gesichtspunkte aus der Literatur nicht objektiv bewertet werden können. Bei der Bewertung der Modellierungssprache sind sowohl objektiv zu bewertende Kriterien als auch ergonomische Gesichtspunkte zu betrachten. Ebenfalls die Benutzbarkeit der Werkzeuge wird von ergonomischen Gesichtspunkten beeinflusst.
Die Arbeit von Wood et al. (1988) gab Hinweise wie die Modellierungssprache zu betrachten sei, von Kitchenham (1996) wurde auf die Bewertung von Werkzeugen eingegangen. Zu Untersuchung dieser beiden Gesichtspunkte wurde ein Fallbeispiel entwickelt. Die dabei gewonnen Erfahrungen, bildeten die Grundlage für die Bewertung.

Wie Abschnitt 3.3 zeigte (siehe auch Übersicht 3.2), überwiegen die Erweiterungen objektorientierter Methoden. Die jüngsten Entwicklungen zur Unterstützung von Multiagentensystemen fanden in diesem Bereich statt. Im folgenden Kapitel werden hauptsächlich solche Erweiterungen charakterisiert und bewertet. Dies geschieht auf Grundlage eines Screenings, dass der Autor dieser Arbeit vorgenommen hat. Wesentlich für die Vorauswahl der im folgenden eingehender betrachteten Methoden, war ihre allgemeine Akzeptanz als anerkannte Methoden für die Realisierung von Agentensystemen, ihre Unterstützung durch Werkzeuge und vor allem ihre Berücksichtigung von BDI-Konzepten. Die gefundene Auswahl an Methoden stimmt mit der Menge der von Dam und Winikoff (2003) untersuchten Methoden überein. Die so als relevant erkannten Methoden werden im Folgenden hinsichtlich ihrer Un-

terstützung der Konzepte und Vorgehensweise betrachtet. Die Ergebnisse dieser Bewertung bestimmen dann die Methoden, die im sechsten Kapitel eingehender untersucht werden. Um angemessen die Modellierungssprache und die pragmatischen Gesichtspunkte betrachten zu können, wird dort ein (kleines) Fallbeispiel mit drei Methoden analysiert. Die entwickelten Modelle und die gesammelten Erfahrungen werden kurz vorgestellt, und unter Berücksichtigung der oben beschriebenen Ergebnisse anderer Autoren wird dann dort eine abschließende Bewertung gegeben.

Kapitel 5

Betrachtung einzelner Techniken und Methoden

In diesem Kapitel werden zuerst einzelne Techniken, später dann einzelne agentenorientierte Methoden, nach den gefundenen Kriterien bewertet. Ihre Benutzung wird kurz charakterisiert, bevor die unterstützten Konzepte und Vorgehensweisen der Methoden untersucht werden.

Tveit (2001) unterscheidet agentenorientierte Entwicklungsansätze zwischen Methoden, die eigene Vorgehensmodelle für agentenorientierte Projekte entwickelten (er nennt diese *high level methodologies*), und denen, die auf Design Ebene (*design level*) einzelne (objektorientierte) Techniken um agentenorientierte Konzepte erweitern. Auch wenn es sich hierbei nicht um Methoden im eigentlichen Sinne, werden zuerst die vorgeschlagenen Techniken der Design Ebene dahingehend untersucht, ob sie bei der Entwicklung von Anwendungen mittels Jadex hilfreich sein können. Danach werden Methoden, wie in Abschnitt 3.1 definiert, betrachtet.

5.1 UML basierte Design–Methoden

Die in diesem Abschnitt besprochenen Ansätze wollen lediglich die grafischen Modelle der von Rumbaugh, Jacobson und Booch (1999) vorgeschlagenen *Unified Modelling Language* (UML) um agentenorientierte Konzepte erweitern. Ziel ist es, bestehende UML-basierten Werkzeuge sowie Erfahrungen, die in der Entwicklung objektorientierter Software gewonnen wurden, weiterhin verwenden zu können. Außerdem würden Entwickler, die bereits mit der UML vertraut sind, sich leichter an eine ähnliche Notation gewöhnen.
Bei der UML handelt es sich nicht um eine Methode in dem Sinne, wie der Begriff in dieser Arbeit verwendet wird. Aber die UML ist eng verbunden mit dem Rational Unified Process. Implizit gehen die Autoren der folgenden Vorschläge davon aus, dass es sich bei Agenten lediglich um "erweiterte Objekte" (siehe Abschnitt 2.2) handelt und daher das architekturzentrierte, iterative Vorgehensmodell des RUP weiterverwendet werden kann. In der UML sind grundsätzlich drei Mechanismen für zukünftige Erweiterungen vorhanden, deren Benutzung die ursprüngliche Syntax und Semantik nicht verändern (siehe Rumbaugh, Jacobson und Booch, 1999). (1) *Constraints* (Beschränkungen), mit denen einzelnen Elementen semantische Be-

schränkungen auferlegt werden können. (2) *Tagged Values* (angehängte Werte), mit denen neue Attribute bestimmten Elementen hinzugefügt werden können. (3) *Stereotypes* (Stereotypen), die es erlauben neue Elemente einzuführen, die Unterklassen (Spezialisierungen) von bereits existierenden Elementen repräsentieren. Diese können zu sog. *Profiles* (Rumbaugh, Jacobson und Booch, 1999:104) zusammengefasst werden. In der *Object Constraints Language* (OCL), einer formalen Sprache, die auf Prädikatenlogik erster Ordnung basiert, werden diese Mechanismen beschrieben. Erweiterungen der UML wurden erfolgreich für eine Reihe von Anwendungsfeldern vorgenommen. So zum Beispiel für die Datenbank–Entwicklung (Rational Software White Paper, 2000), die Modellierung von Webapplikationen (Rational Software White Paper, 1999) und die Modellierung von Geschäftsprozessen (Rational Software White Paper, 2001b).

5.1.1 AUML

Agent UML (AUML)[1] (Odell, Parunak und Bauer, 2000; ist eine Initiative, die die UML weiterverwenden will, um Agentensysteme zu beschreiben. Hierzu soll die UML mit den Notationen anderer agentenorientierter Methoden kombiniert werden. Ein *Modeling Technical Committee* der FIPA ist damit beauftragt, einen Standard für die AUML–Notation zu entwickeln. Die folgenden Notationen werden in Betracht gezogen, um aus ihnen eine einheitliche Notation für Agentensysteme zu entwickeln:

- ADELPHE [Bernon et al., 2002]
- AOR [Wagner, 2003]
- Bric [Ferber, 1995]
- Gaia [Wooldridge, Jennings und Kinny, 2000]
- Aalaadin [Ferber und Gutknecht, 1998]
- MESSAGE [Caire et al., 2001a; 2001b]
- OPM [Sturm, Dori und Shehory , 2003]
- PASSI [Cossentino und Potts, 2002]
- Prometheus [Padgham, 2002; Padgham und Winikoff, 2002]
- Styx [Bush, Cranefeld und Purvis, 2001]
- Tropos [Giunchiglia, Mylopoulos und Perini, 2001]
- UML 2.0 [Object Management Group, 2003]

An diesem Ansatz eine Notation für Agenten zu entwickeln die den KI–Anteilen eines Agentensystems nicht Rechnung trägt, wird auch Kritik geäußert. So schreiben z. B. Caire et al. (2001a):

> ...specifying an objects behaviour in terms of interaction protocols does not make it an agent.

Große Akzeptanz haben die Interaktionsdiagramme der AUML gefunden. Mehrere Methoden, die später charakterisiert werden, haben diese Diagramme übernommen, um den Austausch von Nachrichten zwischen Agenten zu veranschaulichen. Diese, von Odell, Parunak und Bauer (2000) vorgeschlagenen Diagramme basierten auf den Sequenzdiagrammen der UML (Version 1.x).

[1] http://www.auml.org

Die Beziehung von UML zu Agenten

UML und AUML beeinflussen sich gegenseitig. Die UML wird von der *Object Management Group* (OMG) weiterentwickelt. Sie hat im UML 2.0 Standard die Idee der Interaktionsdiagramme übernommen und erweitert. Diese umfangreichere Notation hat die AUML wiederum übernommen, so dass nun für Interaktionen eine einheitliche Darstellung existiert. Abbildung 5.1 zeigt eine Gegenüberstellung der beiden Notationen. Links wird die frühe Form der Interaktionsdiagramme gezeigt, rechts ist eine Interaktion in der aktuellen (A)UML–Notation (nach UML 2.0) dargestellt. Der augenfälligste Unterschied ist die mögliche Einbettung anderer Diagramme in diese Darstellung einer Interaktion. So können Verarbeitungsschritte innerhalb der Kommunikationspartner im Zusammenhang der Kommunikation dargestellt werden. In

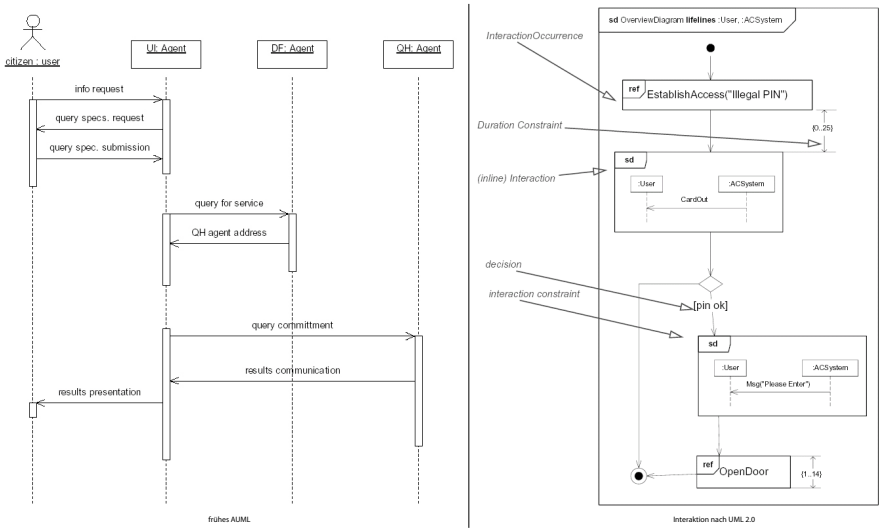


Abbildung 5.1: Interaktionsdiagramme in AUML (linke Seite aus Bresciani et al., 2002; rechte Seite aus Object Management Group, 2003)

der (vorläufigen) UML 2.0[2] Spezifikation (Object Management Group, 2003) wird ein *active object* definiert als:

> An object that may execute its own behavior without requiring method invocation. This is sometimes referred to as "the object having its own thread of control". The points at which an active object responds to communications from other objects are determined solely by the behavior of the active object and not by the invoking object. This implies that an active object is both autonomous and interactive to some degree.

Diese aktiven Objekte sind Instanzen von den sog. aktiven Klassen (*active class*). In der Modellierung werden sie kaum anders dargestellt als herkömmliche Objekte. Wichtig ist lediglich die Unterscheidung in der Semantik. Diese Objekte senden Nachrichten, ohne dazu von anderen Objekten veranlasst worden zu sein.

[2]Diese Spezifikation soll voraussichtlich im Juni 2004 letztendlich angenommen werden

In Kapitel 2.2 wurde auf die Unterschiede zwischen Agenten und Objekten hingewiesen. Diese Definition beschreibt zwar keine Agenten in dem Sinne wie sie hier verstanden werden, aber AUML konzentriert sich auf eine abstrakte Modellierung der Agenten, ohne interne Architekturen (wie z. B. BDI) zu berücksichtigen. Odell (2003) sieht hierin eine Annäherung der agenten- und objektorientierten Entwicklungsansätze.

5.1.2 Weitere Ansätze zur Erweiterung der UML

Yim et al. (2000) haben einen Ansatz vorgeschlagen, der nur die oben beschriebenen Erweiterungsmöglichkeiten der UML Notationen benutzt. Hierfür bieten sie eine Transformation an, die agenten-orientierte in objektorientierte Modellierungsprobleme übersetzt. Modelle nach diesem architekturzentrierten Ansatz gehen von folgenden Voraussetzungen aus. (1) Die elementaren Elemente der Modellierung sind Agenten. Dies bedeutet, dass nicht die interne Struktur von Agenten, ihre mentalen Zustände oder Ähnliches modelliert werden. (2) Das System besteht aus *Components* (Komponenten) und *Connectors* (Verbinder). Agenten nehmen die jeweiligen Rollen als Component oder Connector an. (3) Die Beziehungen zwischen den Agenten werden durch Design Pattern ausgedrückt, um Wiederverwendung zu erreichen. Insbesondere die Beziehungen zwischen Agenten werden in Design Pattern (beschrieben in Abschnitt 5.2; Gamma et al. 1997) übersetzt. Diese Pattern beschreiben dann die Beziehungen zwischen den einzelnen Objektklassen (im Gegensatz zu den üblichen objektorientierten Beziehungen wie Vererbung, etc.). Diese Methode will vor allem die Struktur und das Verhalten groß skalierter MAS modellieren.

Bergenti und Poggi (2000) schlugen 4 neue Diagrammtypen vor, die auf den bekannten UML Notationen basierten. Wie die eben beschriebene Arbeit, bleibt auch dieser Ansatz konform zum UML Metamodel, um von herkömmlichen CASE-Werkzeugen für die UML–Notation verwendet werden zu können. Sie führen lediglich neue Stereotypen ein und benutzen bekannte Diagrammtypen, um Agenten als atomare Einheiten darzustellen. Somit richtet sich ihr Ansatz ebenfalls nur an die Modellierung eines Systems und seiner beteiligten Agenten. Der Aufbau einzelner Agenten wird vernachlässigt, da er nach ihrem Vorschlag objektorientiert modelliert werden soll. Zur Darstellung der Domäne, in der sich die Agenten bewegen, wird ein sog. *ontology diagram* vorgeschlagen. Hierbei handelt es sich um ein abgewandeltes Klassendiagramm, in dem für die Klassen nur *public* Attribute zugelassen sind. Durch diese Felder (in Ontolgie-Werkzeugen oft *Slots* genannt) soll die Struktur der Elemente beschrieben werden. Eine Abwandlung der UML-Implementierungsdiagramme (genannt *architecture diagram*) stellt die Verteilung der Agenten im System dar. Die Agenten werden durch *public*–Methoden beschrieben, die die Aufgaben darstellen, deren Erfüllung bei ihnen angefragt werden kann. Um die Kommunikation zwischen den Agenten darzustellen werden hier abgewandelte Kollaborationsdiagramme (sog. *protocol diagram*) benutzt. An der verbreiteten Darstellung in AUML–Interaktionsdiagrammen wird kritisiert, dass sie nicht UML–konform sind, und daher nicht in herkömmlichen CASE-Werkzeugen erstellt werden können. In diesen Protokolldiagrammen wird von den konkreten Agenten abstrahiert und die Kommunikation zwischen Rollen dargestellt. Diese Rollen werden wiederum in eigenen Stereotypen, die Klassen ersetzen, dargestellt (bezeichnet als *role diagram*).

5.1.3 Übersicht über die Eigenschaften

Tabelle 5.1 fasst die Unterstützung der in Abschnitt 4.3 vorgestellten Kriterien der Konzepte und Vorgehensweise zusammen.[3] Alle beschriebenen Ansätze wollen

	AUML	Weitere Ansätze
Konzepte:		
BDI–Architektur	– –	– –
Autonomie	-	n.v.
Proaktivität	–	n.u.
Capabilities	– –	– –
Events	+	n.u.
Rollen	+	+
Verteilung	++	+
Protokolle	++	+
Nachrichten	+	–
Vorgehensweise:		
Abdeckung der Arbeitsschritte	n.u.	n.u.
Management	n.v.	n.v.
Komplexität	n.v.	n.v.
Eigenschaften des Vorgehens	n.v.	n.v.
Legende - Unterstützung:[3]	– –: sehr schlecht –: schlecht n.v.: nicht verfügbar +: gut +(+): fast sehr gut ++: sehr gut	

Tabelle 5.1: Bewertungsergebnisse: Eigenschaften und Vorgehensweise für UML–basierte Ansätze

Agenten als allgemeine Abstraktion modellieren, im Sinne von *aktiven Objekten*. Der Aufbau dieser Agenten interessiert nicht weiter. Es wird davon ausgegangen, dass die Agenten mit objektorientierten Techniken entwickelt werden. Die speziellen Architekturen der Agenten, beispielsweise der BDI–Aufbau, werden nicht weiter berücksichtigt. Einzig in AUML wird die interne Arbeitsweise einzelner Agenten durch Aktivitäts- und Zustandsdiagramme beschrieben. Eine explizite Beschreibung dessen, was diese Arbeitsweise bewirken soll, bzw. welche Aufgaben ein Agent (bzw. eine Rolle) zu erfüllen hat (Autonomie und Proaktivität), wird jedoch nicht entwickelt. Die anderen beiden beschriebenen Ansätze modellieren Agenten als atomare Einheiten. Folglich werden auch hier Autonomie und Proaktivität nicht berücksichtigt. Events und Nachrichten werden, wie in der UML, zwar modelliert, aber ihre Eigenschaften nicht weiter spezifiziert. Die letzten beiden Ansätze konzentrieren sich auf die Struktur der Systeme und vernachlässigen daher die Modellierung der Events. Die Rollen werden in der AUML, in Interaktions-, Kollaborations- und Aktivitätsdiagrammen verwendet und so ihr Verhalten beschrieben. Bergenti und Poggi verwenden diese auch in den Protokolldiagrammen. Die Verteilung in einem Agentensystem wird sehr anschaulich durch Abwandlungen der UML–Implementationsdiagramme

[3] Zur Zusammenfassung der Ergebnisse wird ein Schema verwendet, welches die Unterschiede in der Unterstützung der betrachteten Kriterien (beschrieben in Abschnitt 4.3) in Relation zueinander darstellt. Die verwendete Skala reicht von einer kompletten Unterstützung (++), bis zur Abwesenheit einer Unterstützung (– –). Sind aus konzeptioneller Sicht gewisse Kriterien nicht anwendbar für die betrachtete Technik/Methode, wurden diese als *nicht verfügbar* (n.v.) bewertet. Die Abstufungen zwischen diesen Extremwerten werden in der jeweiligen Darstellung der Bewertungsergebnisse begründet.

beschrieben. In AUML wird die Mobilität von Agenten durch einen neuen Stereotyp gezeigt. Auch die Protokolle werden, der UML sehr ähnlich, in den bereits erwähnten Interaktionsdiagrammen oder Kollaborationsdiagrammen dargestellt. Die Nachrichten werden in den Diagrammen wie Methodenaufrufe eingezeichnet. Zur Entwicklung wird implizit der UP weiterverwendet, aber für Verwendung in den einzelnen Arbeitsschritten werden keine Hinweise gegeben. Auch das Management eines Projektes wird nicht berücksichtigt. Somit ist die Frage, wie verständlich die Arbeitsweise der Entwicklung ist (Kriterium: Komplexität). Diese ist sehr von der Art des zu entwickelnden Projektes abhängig. Für Projekte, die eine bloße Abstraktion von autonomen Prozessen suchen, wird die herkömmliche Vorgehensweise angemessen sein.
Diese Betrachtungen zeigen, dass diese Ansätze sich darauf konzentrieren die statischen Strukturen eines Systems darstellen. Die Beziehungen der Agenten miteinander werden hauptsächlich durch den Nachrichtenaustausch charakterisiert. Eine zielgerichtete Arbeit der Agenten wird nicht dargestellt. Agenten werden lediglich als autonome, kommunizierende Einheiten beschrieben. Auch die Verteilung der Agenten im System wird dargestellt. Die im Weiteren betrachteten Methoden vernachlässigen letzteres. Die Ansätze konzentrieren sich auf die Gemeinsamkeiten zwischen Objekten und Agenten. Die Anpassungen an die objektorientierte Notation bestehen hauptsächlich darin die Notation zu vereinfachen und so durch das weglassen von Details eine Sicht auf abstraktere Einheiten zu schaffen. Sie berücksichtigen nicht die Existenz von Agentenplattformen, die direkt agentenorientierte Konzepte den Anwednungsentwicklern zur Verfügung stellen. Sie verstehen sich als Aufsatz auf traditionelle Designtechniken, in denen in objektorientierten Sprachen einer Software Eigenschaften eines Agentesystems selbst implementiert werden sollen.

5.2 Design Pattern

Gamma et. al (1997), definieren *Design Pattern* als Beschreibungen von kommunizierenden Objekten und Klassen, um generelle Designprobleme in einem bestimmten Kontext zu lösen. Die Argumente für Design Pattern in der Entwicklung von Agenten sind die gleichen, die den Design Pattern von Gamma et al. so viel Anerkennung eingebracht haben. Insbesondere werden die meisten Agentenplattformen unabhängig voneinander entwickelt, so dass es kaum Wiederverwendung von Design und Komponenten gibt.

Bei der Entwicklung einer Agentenplattform bei IBM wurden zehn verschiedene Design Pattern gefunden, vorgestellt von Aridor und Lange (1998). Die Beschreibung dieser Pattern wird zwar in objektorientierten Begriffen gegeben, was aber nur zweckmäßig ist, da sich objektorientierte Sprachen (insbesondere Java™) bei der Implementation von Agentensystemen durchgesetzt haben. Die Pattern werden in drei Klassen einteilt. Es wird unterschieden zwischen *Travelling Pattern*, *Task Pattern* und *Interaction Pattern*. Die Travelling Pattern beschreiben Mechanismen, die die Mobilität der einzelnen Agenten unterstützen. Da unterschiedliche Plattformen den Anwendungsentwicklern verschiedene Dienste zu Verfügung stellen, werden hier Mechanismen beschrieben, die zur Implementation dieser Dienste genutzt werden können. Die Task Pattern beschäftigen sich mit der Bearbeitung von Aufgaben. Sie zeigen auf, wie Aufgaben von Agenten abgearbeitet, bzw. unterteilt und delegiert werden können. Die Interaction Pattern wiederum beschreiben, nach welchen Mus-

tern Agenten miteinander kommunizieren können. Auch hier werden Mechanismen beschrieben, die die Implementation von Agentenplattformen betreffen.

Kendall et al. (1998) schlagen eine siebenschichtige Architektur für Agenten vor und beschreiben Design Pattern für die einzelnen Schichten. Diese Schichten sind: *mobiliity*, *translation*, *collaboration*, *actions*, *reasoning*, *beliefs* und *sensory*. Dies beschreibt eine generelle Architektur. Nicht alle Agenten bestehen aus allen Schichten, nicht benötigte werden weggelassen, wobei die Richtung der Verarbeitung beibehalten wird (siehe Abbildung 5.2). Nachrichten kommen an der Mobility–Schicht an.

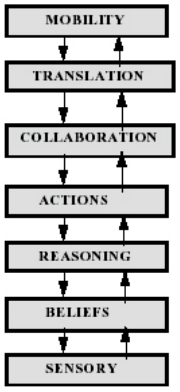


Abbildung 5.2: Schichtenarchitektur für Agenten (aus Kendall et al.,1998)

Die Translation–Schicht übersetzt gegebenenfalls Nachrichten in ein Format, das die höheren Schichten des Agenten verarbeiten können. Die Collaboration–Schicht entscheidet, ob ein Agent eine Nachricht verarbeiten soll. Ist dies der Fall, führt die Actions–Schicht Pläne aus, die von der Reasoning–Schicht ausgewählt wurden. Die Belief- und Sensory–Schichten beinhalten das Modell des Agenten über sich und seine Umwelt. Die unteren drei Schichten (*reasoning*, *beliefs*, *sensory*) bilden das mentale Modell des Agenten. Aspekte dieser Schichten können konzeptioneller anstatt architektonischer Natur sein, insbesondere, wenn aus Gründen der Performance direkte Verbindungen zwischen nicht übereinander liegenden Schichten nötig sind. Die Design Pattern schlagen spezielle Ausprägungen der einzelnen Schichten vor (das mentale Modell wird insgesamt betrachtet), die objektorientiert gestaltet werden, je nachdem welche Art von Verhalten von dem Agenten gewünscht wird.

Die beiden aufgeführten Ansätze ähneln sich, indem sie in gleicher Weise klassifizieren (wie in Tveit, 2001 analysiert). Die Mobility–Schicht und die Translation–Schicht entsprechen den *Travelling Pattern*. Die Collaboration–Schicht entspricht der Klasse der *Interaction Pattern*, und die Actions–Schicht entspricht den *Task Pattern*. Allerdings will der zweite Ansatz alle grundlegenden Arten von *Agent Design Pattern* durch seine Referenzarchitektur klassifizieren.

5.2.1 Übersicht über Eigenschaften

In Tabelle 5.2 ist die Unterstützung der in Abschnitt 4.3 vorgestellten Kriterien, der Konzepte und Vorgehensweise aufgetragen. Aridor und Lange vernachlässigen

	Aridor und Lange	Kendall et al.
Konzepte:		
BDI–Architektur	– –	–
Autonomie	++	++
Proaktivität	n.v.	+
Capabilities	– –	– –
Events	–	(+)
Rollen	++	n.v.
Verteilung	–	n.v.
Protokolle	++	n.v.
Nachrichten	+	++
Vorgehensweise:		
Abdeckung der Arbeitsschritte	1	1
Management	n.v.	n.v.
Komplexität	n.v.	n.v.
Eigenschaften des Vorgehens	n.v.	n.v.
Legende - Unterstützung:	– –: sehr schlecht –: schlecht n.v.: nicht verfügbar +: gut +(+): fast sehr gut ++: sehr gut	

Tabelle 5.2: Bewertungsergebnisse: Eigenschaften und Vorgehensweise für Erweiterungen von Design-Pattern

die mentalistischen Konzepte. Die Architektur von Kendall et al. kennt zwar Pläne und Beliefs, aber die eigentliche BDI–Architektur sowie das Konzept der Capabilities wird in der beschriebenen Form nicht unterstützt. Sie will Agenten allgemein beschreiben. Eine Erweiterung der Reasoning–Schicht wäre aber denkbar, um dies zu tun. Die Autonomie der einzelnen Agenten wird in beiden Ansätzen anschaulich dargestellt. Es wird aufgezeigt, welche Aufgaben die Agenten durch welche Aktionen lösen können und wie sie die Entscheidungen zur Ausführung bestimmter Aktionen treffen. Die Sammlung von Pattern von Aridor und Lange konzentrieren sich oft auf Implementationsaspekte für eine Agentenplattform, daher steht in diesen die Darstellung der Proaktivität nicht zur Frage. Die siebenschichtige Architektur modelliert die Ziele eines Agenten nicht. Aber eine Erweiterung der Reasoning–Schicht um diese wäre denkbar. Dann sollten die Ziele eines Agenten leicht mit der Entscheidungsfindung und den zur Verfügung stehenden Plänen darstellbar sein. Events werden von Kendall et al. zwar nicht detailliert beschrieben, aber sie modellieren eine Schicht für Sensoren, die die Beliefs entsprechend von Veränderugen der Umwelt manipulieren. Dieser Mechanismus, dass die Beliefs eines Agenten automatisch auf einen aktuellen Stand gebracht werden und diese Veränderungen Reaktionen im Agenten bewirken, kann in Jadex nachgebildet werden, ist aber nur ein Teilaspekt des Systems. Daher ist diese Art der Beschreibung von Events zu eingeschränkt. Die Verteilung wird von Aridor und Lange nicht weiter berücksichtigt, stattdessen werden die Kommunikationsbeziehungen zwischen Agenten beschrieben. Wie bereits erwähnt, konzentrieren sich Kendall et al. auf die Beschreibung des Aufbaus und der internen Verarbeitung einzelner Agenten. Also ist dort die Verteilung der Agenten nicht weiter von Belang. Allerdings wird die etwaige Mobilität der Agenten berücksichtigt. Auch die Rollen, die Agenten einnehmen, beschreiben das Verhalten eines Agenten nach außen. Sie werden ebenfalls in dieser Darstellung nicht betrachtet. Da Aridor und Lange von den Beschreibungen von Pattern der Objekt-

orientierung beeinflusst wurden, werden diese Rollen explizit beschrieben. Sie dienen als zentrales Element, um zu definieren, wie die Agenten zusammenarbeiten. Die Protokolle zwischen den Agenten werden bei Aridor und Lange durch die Interaktionsdiagramme der AUML beschrieben. Wie in den objektorienterten Pattern, ist die Kommunikaton zwischen den Agenten ein wichtiger Aspekt des Designs. Kendall et al. konzentrieren sich dagegen auf die Beschreibung der Abläufe innerhalb eines Agenten. Es wird beschrieben, wie eingehende Nachrichten verarbeitet werden und welche Abläufe dazu führen, dass Nachrichten versandt werden. So werden die Protokolle zwischen den Agenten implizit beschrieben. Aus den definierten Abläufen ist ersichtlich (wenn auch nicht offensichtlich), welche Nachrichten wann, aus welchen Gründen, verschickt werden. Die ausgetauschten Nachrichten werden in den Ansätzen nicht als ACL–Nachrichten beschrieben. Auch hier wären natürlich entsprechende Erweiterungen denkbar. Da es sich bei den vorgestellten Entwicklungsansätzen um Vorschläge zum Design handelt, sind die Eigenschaften eines Vorgehensmodelles nicht beschrieben. Implizit wird davon ausgegangen, dass traditionelle Vorgehensmodelle weiterverwendet werden können.

Die von den Autoren beschriebenen Design Pattern sind insgesamt für Jadex ungeeignet. Allerdings ist die Idee, den Anwednungsentwicklern wiederverwendbare Muster von zusammenarbeitenden Agenten an die Hand zu geben, interessant. Die Art der Definition dieser Muster ist stark von der verwendeten Plattform abhängig. Auch für die Design Pattern der Objektorientierung existieren verschiedene Abwandlungen für die einzelnen Implementationssprachen. Das eine Reihe solcher Pattern auch über implementationssprachen hinweg benutzt werden können, ist nur möglich, da ein allgemein anerkanntes Verständnis davon existiert, was objektorientierte Programmiersprachen ausmacht. Von diesem allgemeinen Verständnis ist die Agentenorientierung noch entfernt.
Sollten also Design Pattern für Jadex entwickelt werden, diese wären ein interessantes Hilfsmittel für Anwendungsentwicklung und Lehre (Beschreibungen wiederkehrender Probleme und deren Lösungen würden die Benutzung des Jadex Systems und der Agentenorientierung verdeutlichen), so müssten diese speziell auf die Plattform abgestimmt sein.

5.2.2 Pattern zur Codegenerierung

Es sei auf zwei Methoden hingewiesen, die die Idee von wiederverwendbaren Pattern explizit zur Generierung von Code benutzen. Inspiriert von den Bibliotheken an Design Pattern, die in UML CASE–Werkzeugen zur Generierung von Quellcode eingesetzt werden können, wurden Werkzeuge entwickelt. Diese Ansätze wurden nicht in die obige Bewertung aufgenommen, da in diese Arbeiten die Pattern als Zusätze zu umfassenderen Methoden verstanden werden.

Für die Methode *PASSI*, zu finden in Tabelle 3.1 und Abbildung 3.2, existiert ein Werkzeug (*AgentFactory*)[4], welches aus einer Menge von Design Pattern Quellcode für zwei nicht BDI-basierte Plattformen generiert. Die Methode sowie dieses Werkzeug wurden nicht weiter betrachtet, da die BDI-Architektur nicht unterstützt wird.

Für die Methode *Tropos*, die ausführlich in Abschnitt 5.3.4 betrachtet wird, wurde

[4]zu finden unter: http://mozart.csai.unipa.it/af/index.htm

ein neuartiges Konzept von Pattern, sog. *social pattern* entwickelt, die die sozialen und intentionalen Strukturen zwischen Agenten beschreiben. Das Projekt *SKwyRL* (**S**ocial ar**Ch**itectures for Agent Soft**w**are S**y**stems Enginee**R**ing)[5] definiert in diesen die sozialen (beteiligte Agenten und ihre Intentionen), intentionalen (Aktivitäten der Services), strukturellen (Beliefs, Events und Plane mit denen Services realisiert werden), kommunikativen (Nachrichtenaustausch) und dynamischen (Events und Pläne) Eigenschaften einer Gruppe von Agenten. Die Pattern und das Werkzeug werden von Do et al. (2003a, 2003b) beschrieben.

5.3 Agentenbasierte Vorgehensmodelle

In diesem Kapitel werden Methoden beschrieben, die einen top–down und iterativen Ansatz zur Modellierung und Entwicklung von agentenbasierten Systemen vorschlagen. Im Gegensatz zu den oben genannten Ansätzen enthalten sie ein eigenes Vorgehensmodell. Fünf dieser Methoden werden kurz dargestellt und die unterstützten Konzepte und Vorgehensmodelle bewertet. Es wird in diesem Zusammenhang auch die (soweit vorhandene) Werkzeugunterstützung dargestellt. Die Kriterien, die zur deren Auswahl geführt haben, waren die über sie vorhandene Dokumentation, ihre allgemeine Anerkennung als ernstzunehmende Ansätze und der Grad ihrer Verbreitung. Diese Kriterien wurden von Dam und Winikoff (2003) aufgestellt (sie kamen zu der gleichen Auswahl).

5.3.1 Gaia

Die Gaia Methode wurde von Wooldridge, Jennings und Kinny (2000; 1999) entworfen. Sie unterstützt die Modellierung einzelner Agenten als auch ganzer Agentensysteme bzw. Organisationen von Agenten. Die Beziehungen zwischen den Agenten sind zur Laufzeit statisch, d. h. sie dürfen sich nicht verändern. Dies gilt auch für die Fähigkeiten der Agenten (kein Lernen). Das zu entwickelnde System muss darauf ausgerichtet sein, (globale) Qualitätsmerkmale zu verbessern. Konflikte zwischen Komponenten sind nicht möglich. Ziel ist es die autonome und problemlösende Natur der Agenten, die Interaktionen und die Bildung von Organisationen zu modellieren. Auf systematische Weise soll, ausgehend von den Anforderungen an das System, ein implementierbares Design erstellt werden.

Der Entwicklungsprozess

Die Anforderungsanalyse wird unabhängig vom verwendeten Paradigma für Analyse und Design verstanden. Außerdem will Gaia Entwickler anleiten, die Entwicklung als Prozess des Designs einer Organisation zu sehen. Gaia beschäftigt sich damit, wie eine Gesellschaft von Agenten die gemeinsamen Ziele des Systems löst und was die einzelnen Agenten dazu beitragen müssen. Im Verlauf des Entwurfes werden fünf verschiedene Modelle entwickelt, deren Abhängigkeiten zueinander in Abbildung 5.3 dargestellt sind. Unterschieden wird zwischen den Modellen der *Analyse* und denen des *Designs*. Eine Organisation wird als eine Sammlung von Rollen angesehen. Der erste Schritt der Analyse ist das Auffinden dieser Rollen. Anschließend werden die Interaktionen zwischen den Rollen modelliert. Die Rollen haben die Attribute *responsibilities* (Verantwortlichkeiten), *permissions* (Erlaubnisse/Genehmigungen),

[5] zu finden unter: http://www.isys.ucl.ac.be/skwyrl/

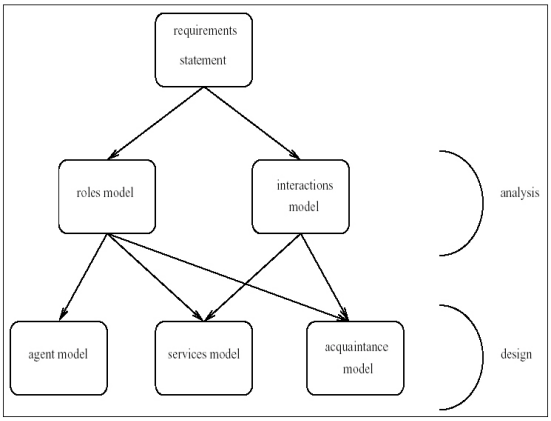


Abbildung 5.3: Gaia: Beziehungen zwischen Modellen (aus Wooldridge, Jennings und Kinny, 2000)

activities (Aktivitäten/Tätigkeiten) und *protocols* (Protokolle). Die Responsibilities bestimmen die Funktionalität der Rolle. Es gibt zwei Arten von Responsibilities. Die *liveness properties* beschreiben Zustände der Umgebung, die von Agenten herbeigeführt werden sollen. Demgegenüber beschreiben die *safety properties* Invarianten. Um diese Aufgaben zu erfüllen, haben Rollen eine Menge von *permissions*. Diese erlauben den Zugriff auf bestimmte Ressourcen (grundsätzlich garantieren diese Genehmigungen aller Art, beschreiben aber meist den Zugriff auf bestimmte Informationen). Die *activities* sind Vorgänge, die die Agenten ohne Beteiligung anderer Agenten ausführen können, um ihre Reponsibilities zu erfüllen. Im Gegensatz dazu werden in den *protocols* Interaktionsprotokolle festgelegt, um die Zusammenarbeit der Rollen festzulegen. Zum Beispiel könnte eine Rolle "Verkäufer" Protokolle für verschiedene Auktionsverfahren (z. B.: dänisch, englisch) haben.
Die Beziehungen der Konzepte der Analysephase sind in Abbildung 5.4 dargestellt. Die Liveness Properties werden durch *liveness expressions*, in einer formale Spra-

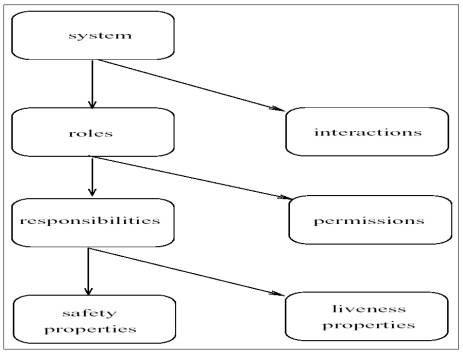


Abbildung 5.4: Gaia: Konzepte der Analyse (aus Wooldridge, Jennings und Kinny, 2000)

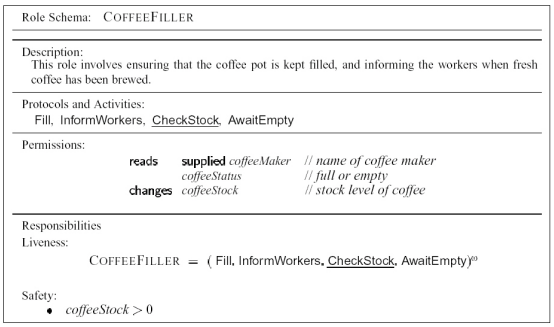
che ausgedrückt. In dieser Sprache werden Activities und Protocols mit Operatoren verbunden, um die Ausführung der Aktivitäten zu beschreiben. Aktivitäten werden unterstrichen dargestellt. So besagt zum Beispiel

$$COFFEEFILLER = (Fill.InformWorkers.\underline{CheckStock}.AwaitEmpty)^{\omega}$$

, dass die Rolle "COFFEEFILLER" die Protokolle "Fill", "InformWorkers", dann die Aktivität "CheckStock" und danach das Protokoll "AwaitEmpty", in dieser Reihenfolge unendlich oft ausführen soll. Die Punkte bezeichnen die serielle Ausführung, das hochgestellte ω steht für unendliche Wiederholung.
Die Safety Properties werden mit Hilfe von Vergleichsoperatoren ausgedrückt, die die Wertebereiche von Variablen (die im Teil der Permissions einer Rolle erlaubt wurden) einschränken. Als einfaches Beispiel drückt

$$coffeeStock > 0$$

aus, dass der Vorrat an Kaffee nicht ausgehen darf.
Das *role model* ist eine Sammlung verschiedener *role schema*, welche die oben aufgeführten Eigenschaften der Rollen beschreiben. In Abbildung 5.5 ist ein solches Schema für einen Agenten zu sehen, der für die Bereitstellung von Kaffee zuständig ist. Es enthält oben dargestellten Formeln für Liveness und Safety Properties.



Role Schema: COFFEEFILLER

Description:
This role involves ensuring that the coffee pot is kept filled, and informing the workers when fresh coffee has been brewed.

Protocols and Activities:
Fill, InformWorkers, CheckStock, AwaitEmpty

Permissions:

reads	supplied *coffeeMaker*	// *name of coffee maker*
	coffeeStatus	// *full or empty*
changes	*coffeeStock*	// *stock level of coffee*

Responsibilities
Liveness:
COFFEEFILLER = (Fill, InformWorkers, CheckStock, AwaitEmpty)$^{\omega}$

Safety:
- *coffeeStock* > 0

Abbildung 5.5: Gaia: Schema für die Rolle COFFEEFILLER (aus Wooldridge, Jennings und Kinny, 2000)

Das *interaction model* besteht aus einer Sammlung von Protokoll–Definitionen, welche die Interaktionen abstrakt definieren. Der Fokus liegt darauf, wer mit wem wozu interagiert. Die Sequenz der Nachrichten wird erst später modelliert. Die Protokoll–Definition besteht aus den folgenden Attributen:

- *purpose* (Absicht/Zweck): Eine knappe Beschreibung der Art der Interaktion (z. B.: "Auftragsvergabe", "Informationsanfrage", etc.)
- *initiator* (Initiator) Die Rolle, die für den Start der Interaktion verantwortlich ist
- *responder* (Antwortende) Die Rolle(n) mit denen der Initiator kommuniziert
- *inputs* Informationen/Ressourcen, über die der Initiator während der Interaktion verfügt

- *outputs* Informationen/Ressourcen, die den Antwortenden mitgeteilt werden, oder die sie mitteilen
- *processing* Eine knappe Beschreibung der Verarbeitung(en), die der Initiator während der Interaktion durchführt.

Ein Beispiel der grafischen Darstellung in Abbildung 5.6. Gezeigt wird das Protokoll

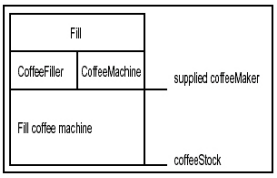


Abbildung 5.6: Gaia: Fill Protokoll (aus Wooldridge, Jennings und Kinny, 2000)

"Fill". "CoffeeFiller" initiiert eine Interaktion mit "CoffeeMachine". "CoffeeMaker" ist *output* an den Initiator und "coffeeStock" ist *input* für den Antwortenden.

Das Ziel des *Design* ist es nicht, die aus der Analyse gewonnenen Modelle so weit zu konkretisieren, dass sie leicht implementierbar sind. Stattdessen werden die Modelle so weit transformiert, dass traditionelle Designtechniken (insbesondere objektorientierte) angewendet werden können. Im Designprozess werden drei Modelle entwickelt.

Im *agent model* werden die *agent types* des Systems und die einzelnen Instanzen dieser Klassen dokumentiert. Sie werden in einem *agent type tree* angeordnet. Die Blätter dieses Baumes repräsentieren die Rollen, die ein Agent des jeweiligen Typs (des Elternknotens) annehmen kann. Hat ein Knoten t_1 zwei Kindelemente t_2 und t_3, dann hat t_1 die Rollen, die die Kindelemente ausmachen. Vererbung spielt in diesem Modell keine Rolle.

Das *sevices model* identifiziert die *services*, die Rollen zugeordnet werden. Ein Service ist eine einzelne Aktivität, die ein Agent ausführen kann. Jeder Activity aus der Analyse wird ein Service korrespondieren, aber nicht umgekehrt. Beschrieben werden sie durch die Attribute: *inputs*, *outputs*, *pre–conditions* und *post–conditions*. Die Services, die ein Agent ausführen kann leiten sich aus den Protcols, Activities und Responsibilities einer Rolle ab. Dieses Modell beschreibt, welche Aktionen eine Rolle asuführen kann.

Das *acquaintance model* wird aus den Roles, Protocols und Agent Models abgeleitet. Es zeigt die Kommunikationsverbindugen zwischen den verschiedenen Agenten. Nur die Verbindungen werden abgebildet. Ziel ist es, etwaige Engpässe zu identifizieren. Sie werden als gerichtete Graphen gezeichnet.

Erweiterungen von Gaia

Gaia ist die erste, speziell für Agentensysteme entwickelte Methode. Es wurden zwei Erweiterungen dieser Methode entwickelt. ROADMAP (Role Oriented Analysis and

Design for Multi-Agent Programming; beschrieben von Juan, Pearce und Sterling, 2002) und SODA (Societies in Open and Distributed Agent spaces; Omicini, 2000). Beide Erweiterungen kritisieren an Gaia die fehlende Modellierung der Umwelt und die mangelnde Darstellung sozialer Strukturen zwischen den einzelnen Agenten.

ROADMAP fügt Gaia vier Erweiterungen hinzu. Es stellt formale Modelle der Umwelt und des Wissens der Agenten, Hierarchien von Rollen, eine explizite Darstellung von sozialen Strukturen und Beziehungen und die Einbeziehung von Veränderungen an diesen dar (in Gaia wird von statischen Strukturen ausgegangen). Eine Übersicht über die einzelnen Modelle dieser Methode zeigt Abbildung 5.7. Die Modelle des Designs bleiben erhalten, der Analyse wurden vier Modelle hinzugefügt (*Use-case Model*, *Environment Model*, *Knowledge Model*, *Protocol Model*).

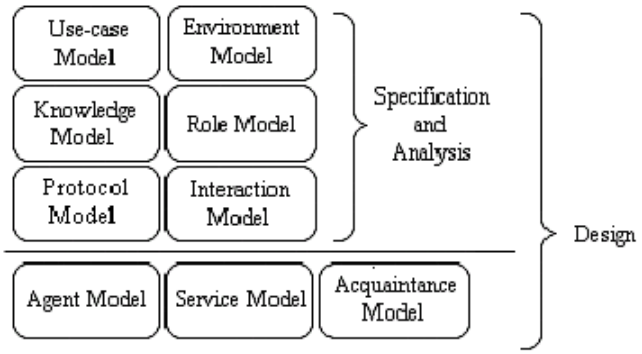


Abbildung 5.7: Übersicht über die Modelle von ROADMAP (nach Juan, Pearce und Sterling, 2002)

SODA konzentriert sich auf die Beschreibung der sozialen Strukturen eines Agentensystems. Auch die Umwelt der Agenten wird in eigenen Modellen dargestellt. Während der Analyse werden nach SODA drei verschiedene Modelle entwickelt:

role model Die Ziele des Systems werden als Aufgaben (tasks), die erfüllt werden müssen, dargestellt. Diese haben Assoziationen zu Rollen und Gruppen.

resource model Die Umgebung der Anwendung wird in verfügbaren Services (verfügbar von abstrakten Elementen der Umwelt) modelliert.

interaction model Die interagierenden Rollen, Gruppen und Ressourcen werden in Protokollen (nach Gaia) beschrieben.

Das Design besteht wiederum aus drei Modellen. Hier werden die Elemente der Analyse auf implementierbare Bestandteile abgebildet:

agent model individuelle und soziale Rollen werden auf Klassen von Agenten abgebildet.

society model Gruppen von Agenten werden auf sog. *societies of agents* abgebildet. Es wird beschrieben, wie die Agenten untereinander koordiniert werden.

environment model Ressourcen werden hier auf Klassen abgebildet und mit topologischen Abstraktionen assoziiert.

5.3.2 MaSE

MaSE steht für *Multiagent Systems Engineering Methodology* (DeLoach, 2001; Wood und DeLoach, 2000). Sie unterstützt (wie der Gaia-Ansatz) sowohl die Entwicklung von einzelnen Agenten, als auch von ganzen Agentensystemen bzw. Organisationen von Agenten. Allerdings gelten Einschränkungen bezüglich der zu modellierenden Systeme. So müssen die Beziehungen der Agenten innerhalb einer Organisation, sowie ihre Fähigkeiten zur Laufzeit statisch sein. Die Interaktionen zwischen Agenten können nur paarweise geführt werden, Multicast ist nicht erlaubt.
Diese Methode betrachtet Agenten nicht aus der Perspektive der Künstlichen Intelligenz, sondern sieht sie als einfache Prozesse, die mit anderen Prozessen interagieren, um gemeinsame Ziele zu erreichen. Agenten werden also nur als Abstraktion angesehen. Durch ihre Zusammenarbeit entstehendes emergentes Verhalten wird nicht weiter betrachtet.

Der Entwicklungsprozess

Die Methode besteht aus sieben Phasen, die in Abbildung 5.8 dargestellt sind. Die Analyse der Anforderungen wird, wie bei Gaia, als unabhängig vom verwendeten Entwicklungsprozess angesehen.
In der ersten Phase *capturing Goals* wird eine Hierarchie der Systemziele erstellt. Hierzu werden die einzelnen Ziele des Systems zuerst, auf Grundlage der Anforderungen an das System, identifiziert und dann nach der Wichtigkeit thematisch geordnet. In der zweiten Phase *Applying Use Cases* werden Anwendungsfälle und Sequenzdiagramme aufgrund der Spezifikation des Systems erstellt. Diese Modelle entsprechen denen aus der UML. Die Anwendungsfälle beschreiben wie sich das System in bestimmten Situationen verhalten soll. Auf deren Grundlage werden dann Sequenzdiagramme erstellt, um den minimalen Nachrichtenaustausch im System festzustellen.
In der dritten Phase *refining the roles* werden Rollen aus den Sequenzdiagrammen abgeleitet, die für die Erreichung der Ziele aus der ersten Phase verantwortlich sind. Zusammen mit den Rollen werden Aufgaben (Tasks) definiert, die beschreiben, wie die einzelnen Ziele, die einer Rolle zugeordnet sind, zu erfüllen sind. Sie werden durch Zustandsdiagramme dargestellt. Die vierte Phase *creating agent classes* bildet die identifizierten Rollen auf Klassen von Agenten ab. Hierzu wird ein *agent class diagram* eingeführt (Abbildung 5.9 zeigt ein Beispiel). Die Agenten werden als Kästen dargestellt, die Verbindungen zwischen diesen Kästen repräsentieren den Kommunikationsaufwand zwischen den Agentenklassen. Um die Performance zu steigern, ist es oft wünschenswert, Rollen mit einem hohen Kommunikationsaufwand zwischen einander, auf eine Klasse von Agenten abzubilden. Die nächsten beiden Phasen bedingen sich gegenseitig. Ihre parallele Durchführung wird vorgeschlagen. Die fünfte Phase, *constructing conversations*, definiert Koordinationsprotokolle in Form von Zustandsdiagrammen. Die Zustände der Agenten während der Kommunikation zwischen zwei Agenten werden hier detailliert beschrieben. In *assembling agent classes*, der sechsten Phase, werden die Architekturen der Agentenklassen definiert. Sie müssen die Kommunikationsmodelle implementieren, die in der vorigen Phase

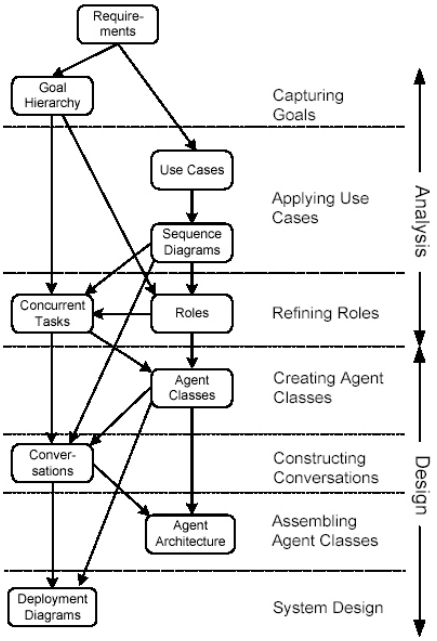


Abbildung 5.8: Abfolge der Arbeitsschritte in MaSE (aus DeLoach, 2001)

erstellt wurden. In der letzten Phase, dem *system design*. Hier werden die Instanzen der Agentenklassen in Implementierungsdiagrammen (der UML) dargestellt.

Werkzeugunterstützung

Das *agentTool*[6] (aktuelle Version: 2.0 Beta) unterstützt die Entwicklung nach MaSE. Es wurde am *Multiagent & Cooperative Robotics Lab* der *Kansas State University* entwickelt. Abbildung 5.10 zeigt die Anwendung bei der Erstellung eines Diagramms, um einen Eindruck von seiner Verwendung zu geben. Hauptsächlich handelt es sich um ein grafisches Werkzeug für die Erstellung der von MaSE verwendeten Diagramme. In Abschnitt 6.2 wird eine Beispielanwendnung nach MaSE entwickelt. Die dortigen Abbildungen wurden mit diesem Werkzeug erstellt. Es ist möglich, die modellierten Conversations auf Deadlocks überprüfen zu lassen. Interessanter Weise kann auch Code für die beiden Frameworks *Agentmom* und *Carolina* generiert werden. Bei Agentmom (beschrieben in DeLoach, 1999) handelt es sich um eine einfache, socketbasierte Kommunikationsinfrastruktur, das Projekt wurde mittlerweile eingestellt. Carolina (beschrieben von Saba und Santos, 2000) ist ein ähnliches Framework zur Entwicklung von Agentensystemen. Der generierte Code besteht hauptsächlich aus den, für die einzelnen Agenten benötigten Java™–Dateien, denen Methodenrümpfe für Versand und Empfang der beschriebenen Nachrichten eingefügt wurden.

[6] http://www.cis.ksu.edu/ sdeloach/ai/projects/agentTool/agentool.htm

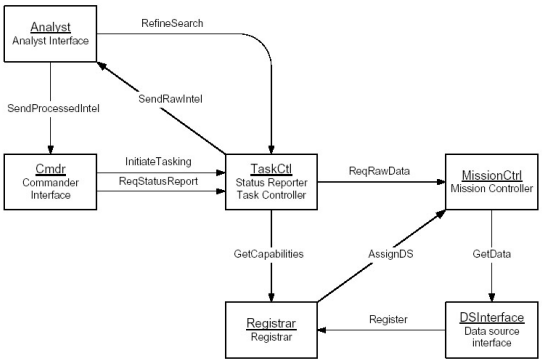


Abbildung 5.9: MaSE - Besipiel eines Agent Class Diagrams (von Wood und DeLoach, 2000)

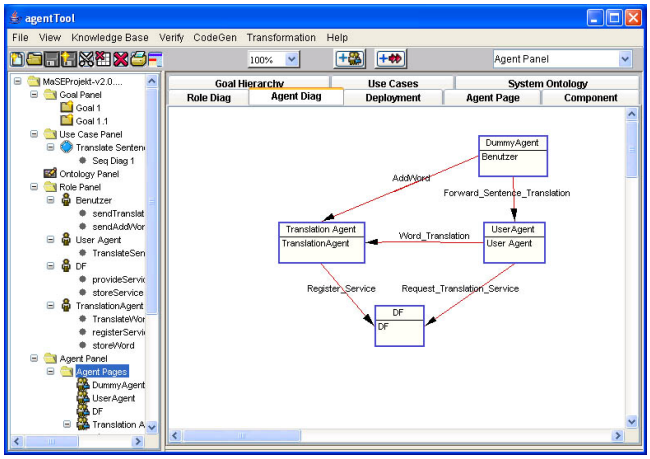


Abbildung 5.10: Die Agent Tool 2.0 Anwendung

5.3.3 MESSAGE/UML

MESSAGE/UML (Methodology for Engineering Systems of Software Agents; Manchmal auch nur als Message bezeichnet) ist ein Ansatz zur grafischen Modellierung (ausführlich beschrieben von Caire et al., 2001a). Diese Methode entstand in einem Projekt der *European Institute for Research and Strategic Studies in Telecommunications* (EURESCOM). Wie der Name vermuten lässt, basiert es auf der UML Notation, erweitert diese aber nicht nur, sondern führt auch neue "Views" und korrespondierende Diagramme ein. Es wird ein eigener Analyseprozess vorgeschlagen, das Vorgehensmodell des UP wurde weitgehend übernommen. Die Autoren verstehen ihren Ansatz als eine Kombination aus (A)UML und Methoden zur agentenorientierten Analyse und Design. Die Idee der Notation nach dem UML Metamodel (insbesondere die Interaktionsdiagramme) wurde von AUML übernommen. Die Analyse und das Design agentenorienterter Systeme wurden bereits in früheren Methoden (Gaia (siehe Abschnitt 5.3.1), MAS-CommonKads (beschrieben von

Iglesias et al., 1998)) unterstützt. Diesen fehlte aber eine entsprechende Notation - sie sind hauptsächlich textbasiert.

Inspiriert von AUML erweitert MESSAGE/UML die UML Modelle. Es ist konform mit dem Metamodell der UML und erweitert dieses um agentenorientierte Konzepte.

Agentenorientierte Konzepte

Es gibt drei Gruppen neuer Konzepte in MESSAGE/UML:

- *Concrete Entities*
- *Activities*
- *Mental State Entities*

Für diese wurden neue Stereotypen (einschließlich neuer Symbole) entwickelt.

Concrete Entities:
Ein **Agent** ist die atomare, autonome Einheit. Sie ist fähig (mehr oder weniger) nützliche Aufgaben auszuführen. Dies wird durch Services ausgedrückt, die Agenten anbieten. Sie sind analog zu den Operationen von Objekten zu verstehen. Sie sind autonom in dem Sinne, dass ihre Aktionen nicht nur von externen Ereignissen oder Interaktionen abhängen, sondern auch von ihrem inneren Zustand. Es wird in diesem Zusammenhang *motivation* genannt und in einem Attribut namens *purpose* abgelegt. Dies beeinflusst, ob und wie ein Agent eine Anfrage eines Service erfüllen wird.

Eine **Organisation** ist eine Gruppe von Agenten, die zusammenarbeiten um einen gemeinsamen Zweck zu erfüllen. Sie sind virtuell, ihre Services werden kollektiv von den Mitgliedern der Gruppe angeboten, der Zweck wird gemeinsam erfüllt. Die interne Struktur wird in (1) Beziehungen der Art Vorgesetzter/Untergebener, und in (2) Koordinationsmechanismen (als Interaktionen) zwischen den Mitgliedern ausgedrückt.

Eine **Role** (Rolle) beschreibt externe Charakteristika eines Agenten in einem bestimmten Kontext. Sie ist analog zu der Beziehung zwischen Interface und Klasse in objektorientierten Systemen zu verstehen. Ein Agent kann mehrere Rollen, und mehrere Agenten können die selbe Rolle annehmen.

Eine **Resource** repräsentiert nicht-autonome Einheiten (z.B. Datenbanken, externe Programme) die Agenten zur Verfügung stehen. Diese werden durch objektorientierte Konzepte modelliert.

Activities:
Ein **Task** (Aufgabe) wird durch Pre- und Postconditions beschrieben. Tasks können auch aus anderen Sub-Tasks zusammengesetzt sein. Die Tasks werden als Zustandsmaschinen angesehen und daher können durch Aktivitätsdiagramme der UML die zeitlichen Abhängigkeiten zwischen diesen untergeordneten Tasks dargestellt werden.

Das Konzept der **Interaction** (Interaktion) ist an die Gaia Methode angelehnt. Mehr als ein Beteiligter nehmen an einer Interaktion teil, um ein gemeinsames Ziel zu verfolgen.

Mental State Entities:
Ein **Goal** (Ziel) assoziiert einen Agenten mit einer Situation. Um die jeweiligen Ziele zu erreichen, versuchen die Agenten die Situation herbeizuführen, die durch das Goal referenziert wird. Ziele können vom Zweck des Agenten abgeleitet sein, so dass er sie seinen ganzen Lebenszyklus lang verfolgt oder von ihm während der Ausführung angenommen und wieder verworfen werden.

Eine **InformationEntity** (Informationseinheit) kapselt Informationen. Sie können von Agenten gehalten oder zu anderen transportiert werden. Der Nachrichtenaustausch zwischen Agenten basiert auf dem Transport der Informationseinheiten.

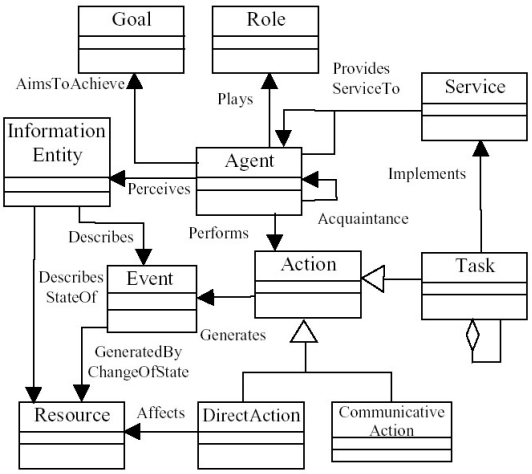


Abbildung 5.11: Message/UML - Konzepte (aus Caire et al., 2001a)

Agentenorientierte Views

MESSAGE/UML schlägt fünf Views mit eigenen Diagramtypen vor. Der **Organisation View (OV)** zeigt die *ConcreteEntities* (Agents, Organisations, Roles, Resources) im System, die Umgebung und eine grobe Übersicht über die bestehenden Beziehungen. Abbildung 5.12 zeigt einen solchen View für eine Organisation (Organization 1), die wiederum aus drei Arten von Organisationen (Organization 2, Organization 3 und Team) bestehen kann. Neben den (Unter–)Organisationen werden auch die enthaltenen Ressourcen, Rollen und konventionelle Klassen angezeigt, die Teil der Organisation sind. Es handelt sich um ein abgewandeltes UML–Klassendiagramm, dem lediglich neue Stereotypen, und mit diesen auch neue Symbole, hinzugefügt wurden.

Der **Goal/Task view (GTV)** zeigt die Goals, Tasks, Situationen und die Abhängigkeiten zwischen ihnen. Goals und Tasks haben beide Attribute vom Typ Situation,

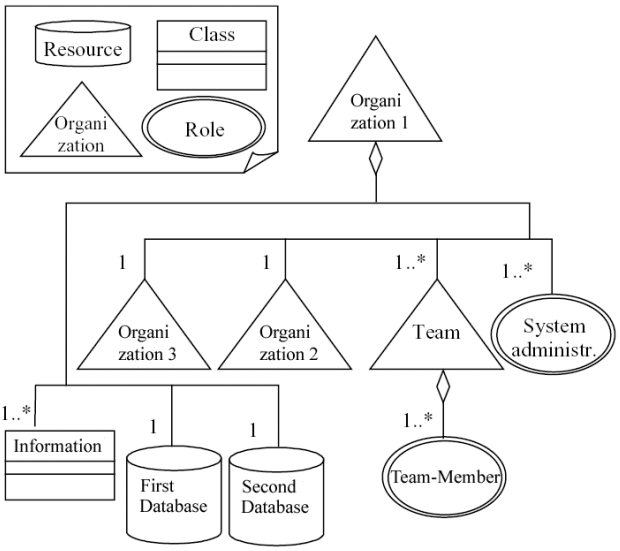


Abbildung 5.12: Ein Organization View (nach Caire et al., 2001a)

so dass in einem Graphen die Auflösung von Goals in untergeordneten Sub-Goals (mit Und/Oder-Beziehungen zueinander) dargestellt und wie Tasks ausgeführt werden können, um Goals zu erreichen. Außerdem werden UML Aktivitätsdiagramme verwendet, um die zeitlichen Abhängigkeiten zu zeigen.

Im **Agent/Role view (AV)** liegt der Fokus auf den individuellen Agenten und Rollen. Die Diagramme werden komplettiert durch schematische Beschreibungen der Agenten und Rollen. Hier werden die jeweiligen Charakteristika (zu erreichende Ziele, zu registrierende Ereignisse, auf welche Ressourcen zugegriffen werden kann, etc.) festgehalten.

Für jede Interaktion zeigt ein **Interaction view (IV)** die Agenten bzw. Rollen, den Initiator der Interaktion, den sog. Motivator (das Goal, für das der Initiator verantwortlich ist), die an der Interaktion Beteiligten, die relevanten Informationen jedes Teilnehmers und die Interaktion beeinflussende Ereignisse. Um die Details eines Interaktionsprotokolles und der Nachrichten, die zwischen Rollen ausgetauscht werden, auszudrücken, werden Sequenzdiagramme der AUML vorgeschlagen.

Die spezifischen Konzepte des jeweiligen Anwendungsbereiches und die Beziehungen zwischen ihnen werden im **Domain view (DV)** dargestellt. Für die Darstellung dieser Ontologie werden Klassendiagramme der UML verwendet.

Der Analyseprozess

Das Ziel der Analyse ist es, ein Modell des Systems und seiner Umgebung zu entwickeln. Es dient als Ausgangspunkt für das Design und dient der Kommunikation zwischen Entwicklern und Kunden. Das endgültige Modell wird schrittweise erstellt.

Die einzelnen Schritte werden als *level* bezeichnet, die durchnummeriert werden. In Level 0 wird das System als eine Sammlung von Organisationen gesehen, die mit Aktoren, Ressourcen und anderen Organisationen interagieren. Aktoren können hierbei sowohl Benutzer also auch andere Agenten sein. Auf dieser Ebene werden *Organisation View* und *Goal/Task View* erstellt. Aus diesen können dann die *Agent/Role Views* und der *Domain View* erstellt werden. Diese Views werden dann zu Rate gezogen, um den *Interaction View* zu entwickeln. Das auf diesem Level entwickelte Modell gibt einen Überblick über das System, seine Umgebung und seine Funktionalität. Der Fokus liegt auf der Identifizierung der Entitäten, Ihren Beziehungen zueinander. In Level 1 werden die Details der internen Struktur und des Verhaltens der einzelnen Einheiten entwickelt. Optional können weitere Level dem Prozess hinzugefügt werden. Diese würden benutzt, um spezifische Aspekte des Systems (z. B. Verteilung, Sicherheit, etc.) zu analysieren.

Da es möglich ist, die einzelnen Views unabhängig voneinander zu erstellen, haben die Entwickler größtmögliche Freiheiten eine Strategie zur Verfeinerung der Modelle zu wählen. Von Caire et al. (2001a) wird ein Fallbeispiel für die Anwendung der Notation und der ersten beiden Level der Analyse gegeben.

Werkzeugunterstützung

MESSAGE/UML erweitert das UML Metamodel. Daher können bestehende UML–Werkzeuge weiterverwendet werden. Es wurden die Anforderungen an ein Modellierungswerkzeug beschrieben (Caire et al., 2001b) und ein Plug–In für das erweiterbare CASE–Werkzeug *MetaEdit+*[7] vorgestellt. Nach Installation des Plug–Ins[8] können die grafischen Modelle erstellt und in HTML oder RTF ausgegeben werden.

5.3.4 Tropos

Die Tropos Methode (Bresciani et al., 2002; Giunchiglia, Mylopoulos und Perini, 2001) legt auf zwei Eigenschaften besonderen Wert. Zuerst auf die sog. "frühe" Analyse der Anforderungen an das System (early requirements engineering), in der die am System Beteiligten und ihre jeweiligen Intentionen identifiziert und analysiert werden. Für Tropos wurde zu diesem Zweck ein Framework zur Modellierung von Anforderung, namens *i**, aufgegriffen, das von Yu (1997) entwickelt wurde. Dessen Modelle und Notationen wurden übernommen. Ziel ist es, ein Verständnis für die Gründe zum Einsatz des zu entwickelnden Systems und die Interaktionen zwischen bereits bestehenden Systemen zu finden. Dies soll später die Analyse der Abhängigkeiten zwischen Teilen des Systems und der Anforderungen an das System (funktionale und vor allem nicht funktionale) unterstützen. Außerdem wird argumentiert, dass ein tieferes Verständnis der Anwendungsdomäne die Auswirkungen von Änderungen am System leichter nachvollziehbar macht.
In i* werden Systeme durch Aktoren, Abhängigkeiten zwischen diesen und deren Ziele beschrieben. Es unterstützt die Anforderungsanalyse jeder Art von Softwaresystemen, die Konzepte und Notationen wurden in Tropos zu einer Entwicklungsmethode für Agentensysteme weiterentwickelt. Zweitens werden agentenbasierte, mentalistische Konzepte (der BDI–Architektur) von Beginn an in allen Phasen der

[7] http://www.metacase.com/mep/
[8] zu finden unter http://www.eurescom.de/ public-webspace/P900-series/P907/index.htm

Entwicklung benutzt. Die Methode wurde im Hinblick auf agentenorientierte Systeme entwickelt, wird von dessen Entwicklern aber auch für objektorientierte Projekte vorgeschlagen. Es wird argumentiert, dass die Erhebung der "frühen" Anforderungen und das daraus entwickelte Verständnis für die Anwendungsdomäne nur förderlich sein kann. Ausgehend von einem Modell des zukünftigen Systems und seiner Umgebung, das inkrementell verfeinert und erweitert wird, soll das implementierbare Design abgeleitet werden.

Die verwendeten Konzepte

Um das verwendete Vokabular aufzuzeigen werden hier die verwendeten Konzepte aufgelistet. Die Begriffe *goal* (Ziel), *plan*, *resource*, *capability*, *role* und *belief*, werden wie im AOSE üblich, verwendet. Der Begriff *Akteur* (actor) repräsentiert eine physische Person, einen Softwareagenten, oder eine Rolle. Abhängigkeiten (*dependency*) zwischen Akteuren (um Ziele/Pläne zu erfüllen, auf eine Ressource zuzugreifen) konzentrieren sich auf Objekte (genannt *dependum*). Der *depender* ist abhängig vom *dependee*.

Der Entwicklungsprozess

Der Entwicklungsprozess nach Tropos gliedert sich in fünf verschiedene Phasen: *Early Requirements*, *Late Requirements*, *Architectural Design*, *Detailed Design* und *Implementation*.

In der Phase der **Early Requirements** sollen die beteiligten Einheiten (Akteure) in der Domäne des Systems (engl. stakeholders) und ihre Ziele identifiziert werden. Das zu entwickelnde System wird noch nicht betrachtet. Bei den Zielen wird unterschieden zwischen *hardgoals* und *softgoals*. Die Hardgoals identifizieren spätere funktionale Anforderungen an das System. Es existieren klare Definitionen und/oder Kriterien, die ihre Erfüllung anzeigen. Für die Softgoals ist es nicht möglich, solche Bedingungen ihrer Erfüllung anzugeben. Sie dienen einer qualitativen Analyse der Anforderungen. Bei einer späteren Definition des Systems können sie in nichtfunktionale Anforderungen übergehen. Ansonsten werden diese beiden Arten von Zielen aber gleich behandelt.
Zwei Arten von Diagrammen werden verwendet. Das *actor diagram* zeigt die Beteiligten und ihre Beziehungen zueinander. Diese Beziehungen werden *social dependencies* genannt und zeigen, wie die Akteure von einander abhängen, um Ziele zu erreichen, Pläne auszuführen und Zugriff auf Ressourcen zu erlangen. Das *goal diagram* zeigt die Analyse der Ziele und Pläne eines einzelnen Agenten. Abbildung 5.13 gibt ein Beispiel für diese Darstellung. Sie zeigt ein System (vom gestrichelten Kreis umschlossen) mit dem zwei Aktoren interagieren. Diese wollen Ziele (als abgerundete Rechtecke eingezeichnet) vom System erfüllen lassen. Im System sind die einzelnen Ziele und Pläne (Rauten) und ihre Dekomposition eingetragen. Einzelne Pläne können von Objekten (Rechtecken) abhängen. Die Richtung der Pfeile gibt hier an, welche Informationen ein Plan benötigt. Die Analyse kann durch drei grundlegende Techniken erfolgen: *Means–end analysis* (identifiziert Pläne, Ressourcen und Softgoals, die dazu beitragen ein Ziel zu erreichen), *contribution analysis* (identifiziert Pläne, die positiv oder negativ auf zur Erfüllung eines Zieles beitragen) und *AND/OR decomposition* (identifiziert eine hierarchische Struktur von Teilzielen durch UND oder ODER Dekomposition).

Die Phase der **Late Requirements** erweitert die Modelle der vorigen Phase. Nun

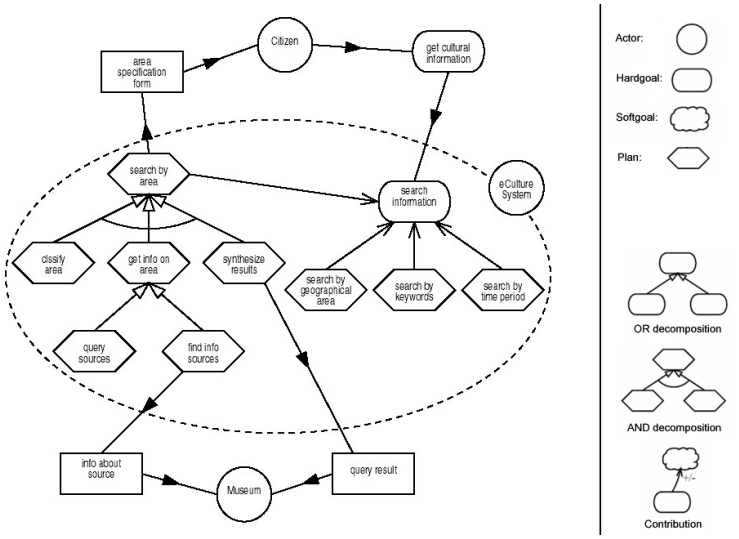


Abbildung 5.13: Tropos - Goal Diagram (aus Bresciani et al., 2002)

wird das zu entwickelnde System im Zusammenspiel mit seiner Umgebung betrachtet. Das System selbst wird mit einem oder mehreren Akteuren modelliert. Abhängigkeiten mit anderen Akteuren dienen lediglich der Erfüllung deren Zielen. Somit definieren sie die Anforderungen (auch die nicht–funktionalen).

In drei definierten Schritten wird das **Architectural Design** erstellt. Die Architektur des Systems wird als eine Organisation von Subsystemen (Akteuren) beschrieben, die durch Daten– und Kontrollfluss miteinander verbunden sind. Zuerst werden neue Akteure eingefügt, um die Architektur des Systems zu bestimmen. Gründe für weitere Akteure könnte die Erfüllung von Teilzielen aus den vorigen Phasen, oder die Erfüllung nicht–funktionaler Anforderungen sein. In den weiteren Schritten werden die Capabilities der Akteure identifiziert und auf Grundlage dessen die Akteure in sog. Agententypen (durch Zusammenfassung einer Konfiguration von Capabilities zu einem Typ) klassifiziert.

Das **Detailed Design** spezifiziert nun die Agenten. In drei Arten von Diagrammen werden die Capabilities, die Pläne und die Interaktionen zwischen den Agenten abgebildet.

Bemerkenswert ist, dass sowohl die Pläne (*plan diagrams*), als auch die Capabilities (*capabilities diagrams*) in UML Aktivitätsdiagrammen beschrieben werden. Abbildung 5.14 zeigt diese beiden Diagramtypen. Das Capability Diagramm (links) zeigt den Kontrollfluss zwischen den enthaltenen Plänen. Da diese auch wieder durch Aktivitätsdiagramme (zum Beispiel der Plan: "evaluate query", rechts) beschrieben werden können, kann der gesamte Kontrollfluss spezifiziert werden. Die Interaktionen zwischen den Agenten werden durch Interaktionsdiagramme der AUML beschrieben (siehe Abschnitt 5.1.1).

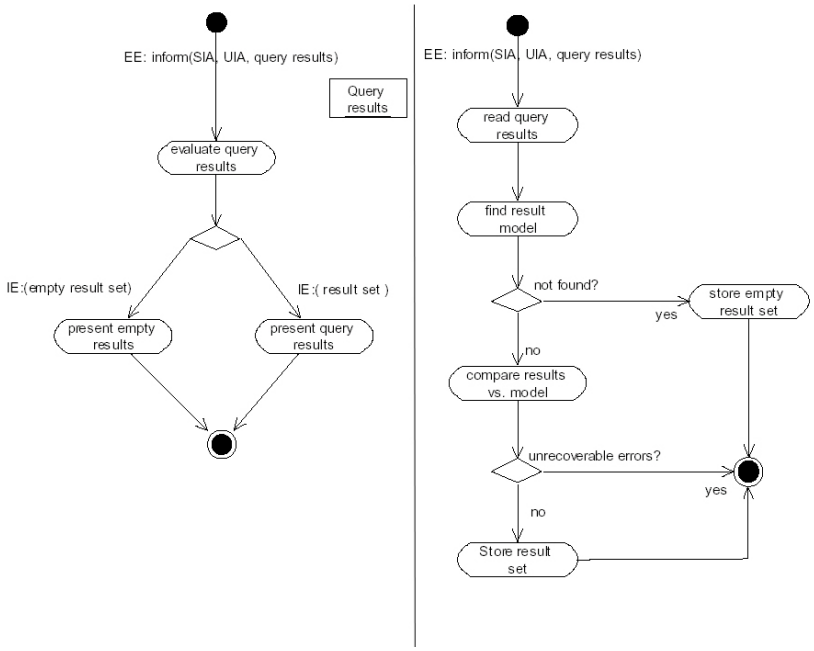


Abbildung 5.14: Tropos - Ein Capablility Diagram (links), und ein Plan Diagram (evaluate query) (beide aus Bresciani et al., 2002)

Werkzeugunterstützung

Es existieren lediglich Werkzeuge, die die i*–Notation[9] unterstützen. In den späteren Arbeitsschritten werden UML–Notationen verwendet. Bemerkenswert ist ein eigenes Projekt zur Code Generierung. In dem Project SKwyRL (**S**ocial ar**Ch**itectures for Agent Soft**W**are S**Y**stems Enginee**R**ing)[10] wurden eine Reihe von sog. *social pattern* definiert. Sie beschreiben generische architekturelle Strukturen für Organisationen von Agenten. Im Gegensatz zu den in Abschnitt 5.2 beschriebenen Pattern (die sich auf die Implmentationsebene der Software konzentrieren), sollen diese Pattern den Design–Arbeitsschritt eines Softwareprojektes unterstützen. Es wurde ein Programm entwickelt, das aus diesen Pattern Quellcode für die JACK Agentenplattform generiert (beschrieben von Do, Kolp, Hoang und Pirotte, 2003). Diese Generierung ist denkbar einfach. Die Pattern enthalten detaillierte Beschreibungen der beteiligten Agenten, die direkt in Quellcode umgesetzt werden.

5.3.5 Prometheus

Prometheus wurde am Royal Melbourne Institute of Technology (RMIT), in enger Zusammenarbeit mit der Firma *Agent Oriented Software Pty. Ltd.*[11] (AOS) entwickelt (Padgham und Winikoff, 2002; Padgham, 2002). Diese Firma vertreibt das JACK–System (eine kommerzielle Agentenplattform; beschrieben von Howden et

[9] T–Tool und OME siehe: http://www.troposproject.org/
[10] http://www.isys.ucl.ac.be/skwyrl/
[11] http://www.agent-software.com

al., 2001), dass die Entwicklung von Agentensystemen mittels einer Agentenplattform und Werkzeugen zur Programmierung unterstützt. Diese Methode wird an dem oben genannten Institut in der Lehre eingesetzt und in Workshops für die Industrie unterrichtet. Ähnlich den bereits erwähnten Methoden wird die Entwicklung von Agentensystemen wird durch die Einführung neuer Modellierungskonzepte, eigener Notationen für entwickelte Modelle und einem Entwicklungsprozess unterstützt.

Der Entwicklungsprozess

Der Entwicklungsprozess in Prometheus besteht aus drei Phasen. Diese werden in einem iterativen Prozess benutzt. Die Artefakte der einzelnen Phasen und ihre Beziehungen untereinander sind in Abbildung 5.15 dargestellt. In der *system specification*

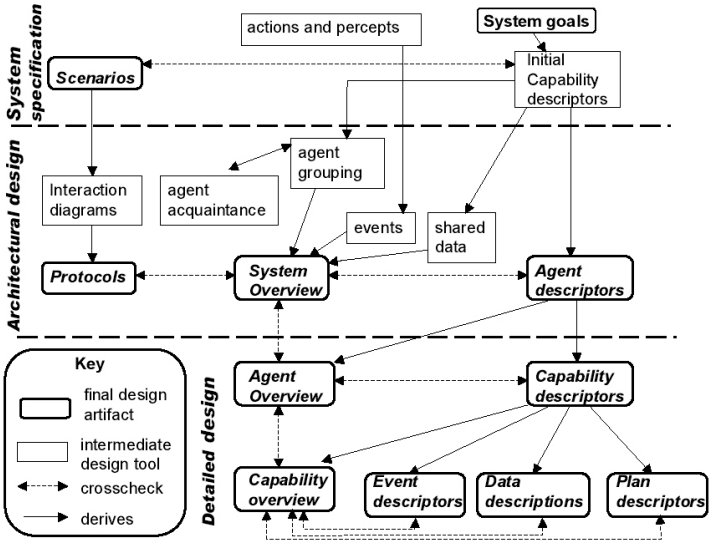


Abbildung 5.15: Prometheus: Übersicht (aus Padgham und Winikoff, 2002)

phase wird als erstes beschrieben, wie das Softwaresystem mit der Umwelt interagiert. Die Mechanismen, die die Umwelt beeinflussen, werden *actions* genannt, *percepts* beschreiben die aus der Umwelt aufgenommenen Informationen. Sie sind von *incidents* (Ereignissen) zu unterscheiden. Die Ereignisse bezeichnen signifikante Vorfälle für das System. Oft ist eine Verarbeitung der erhalten Percepts nötig, um zu entscheiden, ob ein Ereignis für das System generiert werden soll. Parallel zur Identifizierung und Spezifizierung der Actions und Percepts werden die *goals* und *functionalities* des Systems beschrieben. Die Goals beschreiben die Aufgaben, die vom System gelöst werden sollen, und die Functionalities sind Beschreibungen einzelner Funktionen mittels derer dies geschieht. Diese Functionalities werden durch einen *functionality descriptor* beschrieben. Dies ist eine schriftliche Beschreibung, die den Namen, einen kurzen Beschreibungstext, eine Liste von Actions, eine Liste der relevanten Percepts, die verwendeten und generierten Daten (siehe nächste Phase) und eine kurze Beschreibung der Interaktionen mit anderen Functionalities enthält. Um einen umfassenderen Überblick über die Arbeitsweise des geplanten Systems zu

erlangen, werden außerdem *scenarios* auf ähnliche Weise beschrieben. In diesen wird die Abfolge einzelner Arbeitsschritte an beispielhaften Situationen aufgezeigt. Die Beschreibung enthält den Namen (evtl. auch eine Identifikationsnummer), eine kurze Beschreibung, eine optionale Beschreibung in welchem Kontext dieses Szenario auftreten könnte, die *Preconditions* (Vorbedingungen), eine Liste der ausgeführten Verarbeitungsschritte, eine Zusammenfassung der benötigten Informationen und zuletzt eine Liste möglicher Variationen des beschriebenen Szenarien. Die Arbeitsweise soll verdeutlicht, nicht spezifiziert werden.

In der *architectural design phase* werden zahlreiche Diagramme erstellt, um den Aufbau des Systems zu spezifizieren. Die erkannten Functionalities werden zu einzelnen *agent types* gruppiert. Für diese Arten von Agenten wird festgelegt, auf welche Umwelteinflüsse und welche Events sie reagieren müssen und welche Aktionen sie ausführen können, um die Umwelt zu beeinflussen (dies geschieht auch wieder durch textuelle Beschreibungen, die *agent descriptors*, siehe unten). Außerdem werden die Kommunikationswege zwischen den Agenten und etwaige *data repositories* spezifiziert. Alle diese Eigenschaften des Systems werden im *system overview diagram* dargestellt, welches eine zentrale Rolle in der Prometheus–Methode einnimmt. Wie der Name vermuten lässt, soll der Aufbau des Systems anschaulich dargestellt werden. Abbildung 5.16 zeigt ein solches Diagramm mit einer Legende der grafischen Elemente die Prometheus benutzt. In diesem Diagramm werden nur die Wege

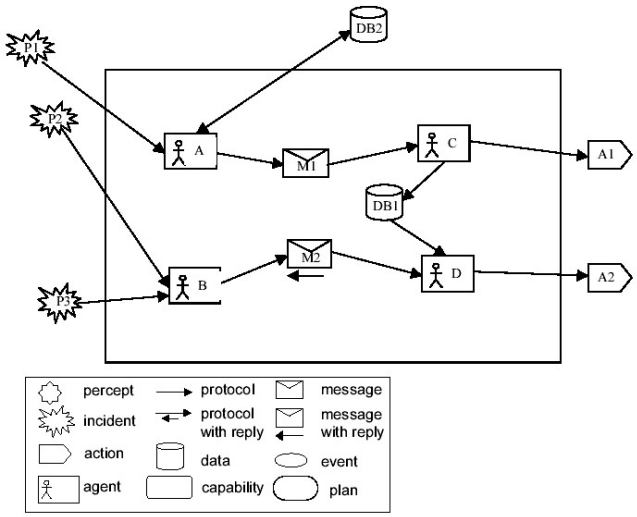


Abbildung 5.16: Prometheus - System Overview Diagram (aus Padgham, 2002)

der Kommunikation, nicht aber ihre zeitliche Abfolge gezeigt. Daher werden die *interaction diagrams* von AUML (siehe Abschnitt 5.1.1) verwendet, um die Abfolge der Nachrichten zwischen Agenten zu zeigen. Die folgenden beiden Diagrammtypen werden benutzt um die Kopplung zwischen den Agententypen zu analysieren. Dazu werden zum einen die gemeinsam benutzten Daten (*data coupling diagram*) und die Assoziationen zwischen den Agenten betrachtet *agent acquaintance diagram*. Im *data coupling diagram* werden die Beziehungen zwischen den identifizierten Daten

(dies meint nicht nur persistente Daten, sondern auch solche die einzelne Functionalities benötigen) und den Functionalities des Systems eingezeichnet. Auf Daten die erzeugt bzw. geschrieben werden, zeigen Pfeile von den schreibenden Functionalities. Pfeile von einzelnen Daten zeigen auf die Functionalities, die sie jeweils verwenden. Im *agent acquaintance diagram* werden Agenten verbunden, die miteinander interagieren. Die Verbindungen zwischen den Agenten sollen verdeutlicht werden. Nach Ansicht der Autoren sind Designs mit wenigen Interaktionen zu bevorzugen. Die Agenten sollen ihre Aufgaben möglichst selbstständig erfüllen. Die *agent descriptors* bilden den Ausgangspunkt für die folgende Phase. In ihnen werden die *agent types*, die im Übersichtsdiagramm dargestellt werden, genauer spezifiziert. Die Beschreibung enthält den Namen, eine knappe Beschreibung der Aufgaben, die Kardinalität (Anzahl der möglicher Agenten dieser Art im System), die *lifetime* (kurze Beschreibung, wann die Agenten erzeugt und wieder zerstört werden), was der Initialisierung und was bei der Zerstörung des Agenten zu tun ist, eine Liste der Funktionalitäten des Agenten, die verwendeten Daten, die produzierten Daten, eine Liste der Ziele, die der Agent zu erfüllen versucht, die Events, auf die der Agent reagiert, die Actions, die der Agent ausführen kann und eine Liste der Agenten mit denen interagiert werden kann.

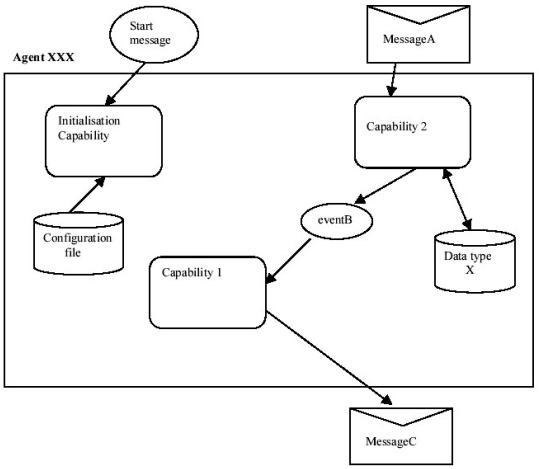


Abbildung 5.17: Prometheus - Agent Overview Diagram (aus Padgham, 2002)

In der *detailed design phase* wird, aufgrund der Spezifikationen, die interne Struktur der einzelnen Arten von Agenten entwickelt. Die Begriffe, die in Prometheus benutzt werden, um die interne Struktur auszudrücken, orientieren sich an der BDI Architektur (wie in Abschnitt 2.3 beschrieben). Agenten sind plan–basiert, d. h. sie benutzen eine Menge vom Benutzer erstellter Pläne. In dieser Phase werden die Capabilities (beschrieben in Abschnitt 2.3), internen Events, Pläne, und detaillierte Datenstrukturen entworfen. Zuerst werden die Capabilities definiert, sie können wiederum Capabilities enthalten und werden durch *capability descriptors* und *capability diagrams* beschrieben.
Die *capability descriptors* beschreiben die Interaktionen mit anderen *capabilities*, die benutzten Daten und ihre Benutzungsschnittstelle. Der Deskriptor enthält den Na-

men, die Listen der Capabilities, mit denen interagiert wird oder die die Capability enthält, die Daten, die gelesen und geschrieben werden und Listen der Events, die als Input dienen oder die von dieser generiert werden. In *capability diagrams* wird der hierarchische Aufbau einer Capability gezeigt. Sie kann sich aus anderen Capabilities zusammensetzen und zeigt auf unterster Ebene die Pläne mit den Events, die diese verbinden. Die einzelnen Elemente, die bisher verwendet wurden, werden wiederum durch die *plan descriptors*, *event descriptors* und *data descriptors* beschrieben. Hier werden die Details spezifiziert, die zur Implementation nötig sind. Diese sind abhängig von der verwendeten Plattform, mit der das Agentensystem realisiert werden soll. Zusammen mit den Definitionen der Bestandteile der Agenten gibt das *agent overview diagram* (Abbilung 5.17) einen Überblick über einzelne Agenten. Es zeigt die vorhandenen Capabilities, den Kontrollfluss zwischen diesen und die Daten innerhalb des Agenten.

Die Unterstützung durch Werkzeuge

Die kommerzielle Agentenplattform JACK beinhaltet die sog. *Jack Development Environment* (JDE). Diese stellt ein grafisches Interface zur Erstellung von Agenten mittels JACK dar. Es werden die Modelle des Arbeitsschrittes des Detailed Design unterstützt und aus ihnen ausführbarer Code erzeugt.
Das *Prometheus Design Tool*[12] (PDT; Version 1.1) wurde am RMIT entwickelt und ist per *Java Web Start*^{TM} frei verfügbar. Es handelt sich um ein CASE–Werkzeug zur Erstellung der Diagramme und Deskriptoren (Dargestellt in Abbildung 5.18). Es unterstützt den gesamten Entwicklungsprozess und kann die Konsistenz der Mo-

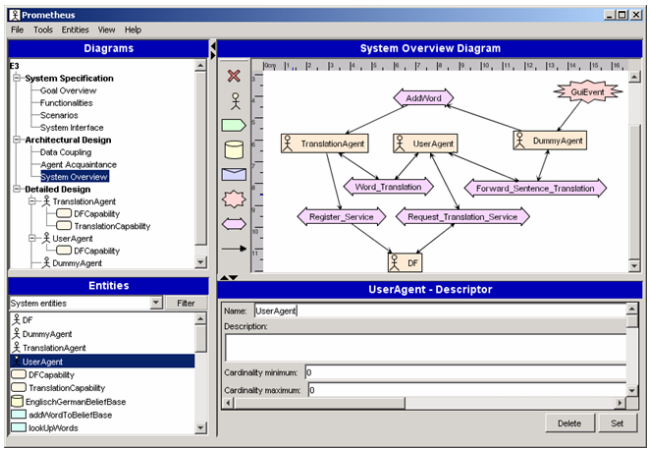


Abbildung 5.18: Das Prometheus Design Tool

delle überprüfen (eine detaillierte Liste der Überprüfungen wird von Padgham und Winikoff (2002) angegeben). Im Gegensatz zur JDE wird kein Code erzeugt.
In beiden Werkzeugen werden die Modelle erstellt, indem die Diagramme gezeichnet, und den Artefakten entsprechende Formulare (für die Deskriptoren) ausgefüllt werden. Im PDT werden hierzu zuerst die Elemente des Systems dem Projekt hinzugefügt. Diese Elemente können dann in den verschiedenen Diagrammen benutzt

[12] http://www.cs.rmit.edu.au/agents/pdt/

und so von anderen Elemente referenziert werden. Die Diagramme in Abschnitt 6.4 wurden mit dem PDT erstellt.

5.3.6 Übersicht über die Eigenschaften der Methoden

In Tabelle 5.3 werden die Methoden, nach den in Abschnitt 4.3 vorgestellten Kriterien der Konzepte und Vorgehensweise, einander gegenübergestellt. Alle fünf eben be-

	Gaia	MaSE	MESSAGE	Tropos	Prometheus
Konzepte:					
BDI–Architektur	+	+	+	++	++
Autonomie	+(+)	++	++	+(+)	++
Proaktivität	+	+	+	++	+(+)
Capabilities	– –	– –	– –	++	++
Events	–	+	+	+	++
Rollen	++	++	++	+	– –
Verteilung	– –	++	– –	– –	– –
Protokolle	+	++	+	++	++
Nachrichten	– –	+	– –	+	++
Vorgehensweise:					
Abdeckung der Arbeitsschritte	2	3	2,5	4	4
Management	n.v.	n.v.	n.v.	n.v.	n.v.
Komplexität	++	++	++	++	++
Eigenschaften des Vorgehens	für alle iterativ und Top-Down				
Legende - Unterstützung:	– –: sehr schlecht –: schlecht n.v.: nicht verfügbar +: gut +(+): fast sehr gut ++: sehr gut				

Tabelle 5.3: Bewertungsergebnisse: Eigenschaften und Vorgehensweise für agentenbasierte Methoden

trachteten Methoden entwickeln das System bzw. die Agenten eines Systems ausgehend von den identifizierten Zielen. Allerdings wird die eigentliche BDI–Architektur in dem Sinne, wie sie von PRS–artigen Systemen benutzt wird, nur von Tropos und Prometheus unterstützt. In Gaia werden die Aufgaben und das Verhalten eines Agenten lediglich formal spezifiziert. Diese Spezifikation kann dann von Entwicklern auch in eine Modellierung in BDI–Strukturen weiter umgesetzt werden, aber die Architektur an sich wird nicht unterstützt. MESSAGE/UML unterscheidet sich in dieser Hinsicht nicht von Gaia.
In MaSE werden die Agenten als Abstraktion für autonome Prozesse angesehen. In der Phase *Assembling Agent Classes* wird zwar eine BDI-Modellierung der Agenten als möglich vorgeschlagen (durch abgewandelte Klassendiagramme), aber dies ist unbefriedigend gegenüber den Methoden Tropos und Prometheus. Diese Methoden verwenden während der gesamten Entwicklung BDI-Konzepte. Nur in Tropos und Prometheus wird die Strukturierung der Agenten durch Capabilities unterstützt.
Die Autonomie (als zentrales Element von Agenten im Allgemeinen) wird von allen Methoden ausdrucksstark dargestellt. Es wurde gezeigt, wie Gaia die Aufgaben eines Agenten formal beschreibt. Hierbei wird nicht dargestellt, wie im Agenten Entscheidungen getroffen werden. Tropos identifiziert lediglich die Ziele, die ein Agent zu erfüllen sucht. Hierfür steht ihm ein Repertoire an Plänen zur Verfügung. Die Autonomie wird also lediglich durch die Assoziation von Agenten mit ihren

Zielen ausgedrückt. Zusätzlich werden hier auch die Abhängigkeiten der Agenten voneinander betrachtet. Die Autonomie wird in MESSAGE/UML und MaSE durch die *Tasks* ausgedrückt, die ein Agent eigenverantwortlich ausführen kann. Prometheus unterstützt ein ähnliches Konzept. Hier stehen den Agenten *Functionalities* zur Verfügung, um ihre Ziele zu erreichen.
Die Ausdrucksstärke der Proaktivität ist eng verbunden mit den BDI-Konzepten. Nur wenn diese durchgängig benutzt werden, lassen sich aus den entwickelten Modellen die proaktiven Eigenschaften der Agenten erkennen. Hier sind ebenfalls Tropos und Prometheus im Vorteil. In MaSE werden sehr deutlich (im Rollen–Diagramm) die einzelnen *Tasks* dargestellt, die ein Agent ausführen kann. Welche Ziele sie erfüllen (sollen) ist allerdings nicht direkt ersichtlich. In MESSAGE/UML werden den Agenten Services zugeordnet, die wiederum Tasks zu deren Erfüllung referenzieren. Dies schafft eine abstraktere Sicht auf die Agenten. Zum Design, welche Ziele ein Agent erfüllen kann, wird wenig beigetragen. In Prometheus werden den sog. *Functionalities* (aus denen die Agenten abgeleitet werden) zu erfüllende Ziele in den Deskriptoren zugeordnet. Dies ist eine anschauliche Darstellung. In Tropos hingegen, wird in den Goal–Diagrammen gezeigt, welche Pläne eines Agenten in welcher Weise (*Means–ends analysis, contribution, AND/OR decomposition*, dargestellt in Abschnitt 5.3.4) zur Erfüllung einzelner Ziele beitragen.
Nur in Prometheus werden Events explizit modelliert. Es wird sogar zwischen *percepts* (erkannten Änderungen an der Umwelt, durch Sensoren) und den daraus für das System relevanten Ereignissen (incidents) unterschieden. Allerdings sind Beschreibungen von Events innerhalb eines Agenten (in Jadex *internal Event* genannt) nicht möglich. In Gaia werden Events lediglich durch erfüllte oder unerfüllte Bedingungen ausgedrückt. Die anderen Methoden beschreiben Ereignisse nur implizit in den UML–basierten Modellen des Designs.
Rollen als Strukturierungskonzept werden in allen Methoden außer in Prometheus unterstützt. Die Methoden verwenden Rollen als Mittel zur Identifikation von Agentenklassen. Diese Klassen ergeben sich aus der Zusammenfassung mehrerer Rollen, die konzeptionell zusammen gehören. In Prometheus hingegen werden Agenten nicht aus diesen Rollen, sondern explizit aus den Aufgaben (Functionalities), die sie erfüllen können, abgeleitet. Agenten werden also als eine Sammlung von Funktionen verstanden. Tropos unterscheidet in der Modellierung nicht zwischen Benutzern, Rollen und (Software–)Agenten. Dies ist konzeptionell korrekt, da (1) die Agenten anderen Akteuren gleichberechtigt agieren sollen, und (2) alle Arten von Akteuren Rollen erfüllen (bzw. annehmen) können, was eine Unterscheidung hinfällig macht. Allerdings führt diese Vereinfachung der Modellierung in komplexen Anwendungsgebieten zu Unübersichtlichkeit.
Die Verteilung und die Mobilität der Agenten innerhalb eines Systems wird kaum dargestellt. Lediglich MaSE verwendet abgewandelte UML–Deploymentdiagramme, um die Verteilung von Agenten auf einzelne Plattformen darzustellen. Die anderen Methoden stellen lediglich die Kommunikationsbeziehungen zwischen den Agenten dar (sog. *Acquaintance* Modelle).
Die Kommunikationsprotokolle zwischen Agenten werden in Tropos und Prometheus durch die Sequenz der übermittelten Nachrichten beschrieben. In Gaia wird die Absicht einer Kommunikation zwischen Agenten beschrieben, sowie die hierzu von den Agenten benötigten Informationen. MESSAGE/UML drückt diese Beschreibung lediglich grafisch aus. In MaSE wird besonderer Wert auf die genaue Modellierung der Agenten gelegt. Beide Seiten der Kommunikation werden durch Zustandsdiagramme der UML Notation beschrieben. Es wird hier der Nachrichtenaustausch

zusammen mit den Zuständen der Agenten, die sie nach Nachrichtenempfang bzw. Nachrichtenversand einnehmen, gezeigt.
In Gaia und MESSAGE/UML werden die Eigenschaften von Nachrichten nicht explizit beschrieben. Auch in Tropos spielen sie eine untergeordnete Rolle. Dort werden sie nur in den Interaktionsdiagrammen (die Notation der AUML wird benutzt) verwendet, um die Protokolle zu beschreiben. Ähnlich wird in MaSE verfahren. Durch die genaue Beschreibung der Protokolle wird dargestellt, wann Nachrichten versandt werden, nicht ihr Inhalt. Als einzige Methode bemüht sich Prometheus durch Descriptoren die implementationsspezifischen Eigenschaften von Nachrichten zu beschreiben. Diese Beschreibungen orientieren sich aber leider nicht an FIPA ACL–Nachrichten.
Bresciani et al. (2002) stellten bei ihrer Beschreibung der Tropos–Methode die Abdeckung von Arbeitsschritten in einem Balkendiagramm gegenüber. In Anlehnung an diese Darstellung wird in Abbildung 5.19 die Abdeckung der Arbeitsschritte des UP dargestellt. Zum Vergleich sind die Methoden aus Kapitel 5.3 und die AUML–Notation aufgeführt. AUML konzentriert sich nur auf das Design eines Systems. Gaia richtet sich an die Erhebung der Anforderungen und der Analyse des Systems. Das Design wird nicht so weit unterstützt wie in MESSAGE/UML. Die weiteren Methoden unterstützen die Prozesse bis zur Implementation in ähnlicher Weise. MaSE erwähnt auch das *Deployment* der Anwendung als eigenen Arbeitsschritt. Tropos zeichnet sich dadurch aus, dass die Analyse der Anforderungen weiter gefasst wird (sie beginnt bereits mit der Analyse des Problembereiches). In Prometheus wird (Poutakidis, Padgham und Winikoff, 2002) die Unterstützung von Tests mittels Designartefakten diskutiert (Poutakidis Padgham Winikoff, 2002). Eine Werkzeugunterstützung, die Prometheus–Diagramme verwendet, um die Kommunikation zwischen den Agenten zu überprüfen, ist angedacht.
Auf das Management eines Softwareprojektes wird in allen Methoden kein Bezug genommen. Es werden meist lediglich Heuristiken zum Entwurf gegeben.
In allen hier betrachteten Methoden scheint die Abfolge der vorgeschlagenen Ar-

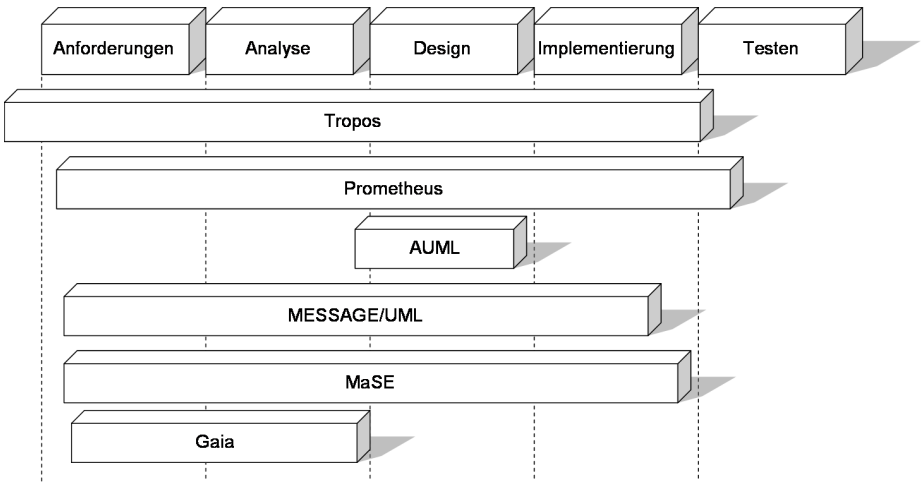


Abbildung 5.19: Die Methoden im Vergleich zum UP (in Anlehnung an Bresciani et al., 2002)

beitsschritte sinnvoll und verständlich. Dies verwundert nicht, da hier eine Auswahl von allgemein anerkannten Methoden betrachtet werden, die über gewisse Zeiträume weiterentwickelt wurden. Alle Methoden können iterativ verwendet werden, es wird in allen top–down, von Ermittlung der Anforderungen (Identifikation der Ziele) hin zur Identifikation und Beschreibung der einzelnen Agenten, entwickelt.

5.4 Zusammenfassung der Bewertungsergebnisse

Die Betrachtung der Ansätze zur Erweiterung der UML[13] und des Einsatzes von Design Pattern[14] haben gezeigt, dass diese Mittel nicht ausreichen, um das agentenorientierte Paradigma ausreichend zu beschreiben. Die Unterschiede zwischen einem objektorientiertem System und einem Agentensystem sind zu groß. Die Modelle des Designs können nicht einfach erweitert werden, um diese Unterschiede handhabbar zu machen. Das Jadex System implementiert eine Reihe von agentenorientierten Konzepten, die in diesen Darstellungsarten nicht berücksichtigt werden.

Die Einschränkungen die Gaia einem zu entwickelnden System auferlegt[15] sind nicht tragbar für das Jadex System. Außerdem geht die Modellierung dieser Methode nicht weit genug. Sie will lediglich auf die Anwendung traditioneller Designtechniken für die einzelnen Agenten vorbereiten. Da in der Implementierung eines Systems mittels Jadex aber nur noch die BDI–Struktur eines Agenten spezifiziert werden soll, und als einziger Code die Pläne der Agenten in Java™ausformuliert werden müssen, ist eine ganzheitliche Unterstützung der Entwicklung mittels dieses Paradigmas wünschenswert.

MESSAGE/UML fügt der Darstellung von Agentensystemen wenig hinzu, was nicht auch in Gaia spezifiziert wird. Es konzentriert sich im Wesentlichen darauf, eine Organisation von Agenten zu beschreiben sowie die Ziele die diese Organisation durch Teilziele und Aktivitäten erfüllen kann.[16] Das Design eines einzelnen Agenten wird hierbei allerdings kaum berücksichtigt. Es versucht, wie die oben genannten Erweiterungen der UML, mit dem UML-Metamodel konform zu bleiben. Dies führt dazu, dass die Ausdrucksstärke im Bezug auf die Modellierung für Agentensysteme eingeschränkt ist. Daher ist es hauptsächlich für die Analyse eines (agentenbesierten–) Softwareprojektes geeignet. Für das Design der konkreten Agenten bieten andere Methoden detailliertere Hilfsmittel.

Daher scheinen die drei verbliebenen Methoden am geeignetsten für die Zusammenarbeit mit dem Jadex System zu sein.

In MaSE wird in besonderer Weise die Modellierung des Kontrollflusses innerhalb eines Agenten mit der Kommunikation verbunden.[17] Besonderer Wert wird auch auf die Darstellung der Funktionen gelegt, die ein Agent (vielmehr eine Rolle) ausführen

[13] siehe Abschnitt 5.1.3
[14] siehe Abschnitt 5.2.1
[15] vorgestellt in Abschnitt 5.3.1
[16] dargestellt in Abschnitt 5.3.3
[17] dargestellt in Abschnitt 5.3.2

kann. Die Agenten werden hauptsächlich als autonome Prozesse verstanden. Was eine starke Einschränkung für die Modellierung darstellt. Zentrales Element ist die BDI–Architektur. In MaSE werden aber die BDI-Konzepte nicht während der gesamten Entwicklung verwendet. Stattdessen wird vorgeschlagen, im Arbeitsschritt des *Assembling Agent Classes* diese Architektur nachträglich in die identifizierten Agenten einzubetten.

Tropos stellt ein Softwaresystem als eine Organisation von Akteuren mit Interessen und Abhängigkeiten untereinander dar. Diese Sicht ist als Metapher für BDI–Agenten besonders gut geeignet.[18] Allerdings wird in den späteren Arbeitsschritten des Designs ebenfalls hauptsächlich der Kontrollfluß durch die Capabilities und Pläne eines Agenten beschrieben.

In Prometheus wird besonders detailliert der Aufbau einzelner Agenten beschrieben. In dieser Methode werden am umfassendsten in Jadex verwendete Konzepte (wie Capabilities, BeliefBases, etc.) berücksichtigt.[19] Die Identifikation der Agenten ist ungewöhnlich. Leider wird hierzu nicht das Konzept der Rollen verwendet. Im Gegensatz zu Tropos wird im Design der Agenten auf eine Darstellung des Kontrollflusses gänzlich verzichtet. Es wird nur der Nachrichtenfluß dargestellt, dieser aber auch innerhalb der Agenten.

[18] dargestellt in Abschnitt 5.3.4
[19] dargestellt in Abschnitt 5.3.5

Kapitel 6

Anwendungsbeispiele

In Abschnitt 5.3 wurden fünf Entwicklungsmethoden für Multiagentensysteme vorgestellt. Die Auswahl dieser Methoden aus dem breiten Spektrum der Entwicklungsmethoden für Agentensysteme waren hauptsächlich begründet durch die unterstützung der BDI–Architektur, die vorhandene Dokumentation und ihre Verbreitung (z. B. ob sie nicht nur von den eigenen Entwicklern benutzt werden). Im vorigen Kapitel wurde eine qualitative Bewertung dieser Vorauswahl vorgenommen, dies führte zur Auswahl von drei Methoden zur weiteren Betrachtung. In den folgenden Abschnitten werden nun diese drei Methoden benutzt, um eine kleine Anwendung zu modellieren. Dies soll ein tieferes Verständnis für die Eigenschaften der jeweiligen Methode schaffen. Die gesammelten Erfahrungen dienen dann als Grundlage, um die Methoden nach den Bewertungskriterien der Modellierungssprache und pragmatischen Gesichtspunkten zu bewerten.

6.1 Aufgabenstellung

In einem Tutorial (Braubach und Pokahr, 2003) werden anhand kleiner, überschaubarer Beispiele die grundlegenden Konzepte des Jadex Systems vorgestellt. Die Studenten der Universität Hamburg benutzen es, um sich mit dem System vertraut zu machen.
In den folgenden Abschnitten werden die verschiedenen Methoden auf das letzte (und komplexeste) Beispielprogramm des Tutorials angewendet. In diesem Programm kommunizieren vier Agenten miteinander. Ziel des Systems ist die Übersetzung von Sätzen aus dem Englischen ins Deutsche[1]. Abbildung 6.1 zeigt die Arbeitsweise des Systems. Der Benutzer des Systems sendet über einen sog. *Dummy–Agenten* (wird von Jade gestellt) Anfragen zur Übersetzung eines Satzes an einen *User Agenten*. Dieser wiederum sendet daraufhin Anfragen zur Übersetzung der einzelnen Wörter an einen *Translation Agent*. Um dies tun zu können, muss zuerst die Adresse des vom *Translation Agent* angebotenen Service (zur Übersetzung von Wörtern), vom DF erfragt werden. Bei seinem Start wird der *Translation Agent* also

[1]Der geneigte Leser wird feststellen, dass dies kaum eine geeignete Problemstellung für ein Multiagentensystem ist. Die Agenten im Tutorial werden schrittweise erweitert - ein möglichst einfaches Beispiel, in dem agentenbasierte Konzepte eingeführt werden hat sich bewährt. Dieses Beispiel ist in diesem Zusammenhang besonders geeignet, da es, in einfacher Weise, die wichtigsten Jadex–Konzepte vorstellt.

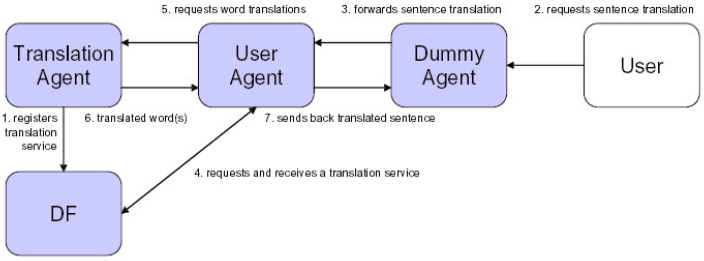


Abbildung 6.1: Arbeitsweise des Beispielsystems (aus Braubach und Pokahr, 2003)

seinen Service dem DF bekannt machen. In Jade ist der DF als Agent realisiert. Dessen Funktionalität kann von Agenten mittels einer eigenen Capability (DFCapabliy - in Jadex enthalten) benutzt werden. Der *Translation Agent* benutzt eine Beliefbase, um Wörter nachschlagen zu können. Sie enthält die englischen Wörter mit ihrer deutschen Übersetzung. Um neue Wörter in die Beliefbase einfügen zu können, soll der Agent auch auf entsprechende Nachrichten reagieren können. Es ist vorgesehen, dass diese Nachrichten vom Benutzer des Systems (Über den *Dummy Agenten*) direkt an den *Translation Agent* geschickt werden.

6.2 Entwicklung nach MaSE

Die Entwicklung nach MaSE wurde in Kapitel 5.3.2 charakterisiert. Eine Übersicht über die einzelnen Phasen, in denen im Entwicklungsprozess nach MaSE unterschieden wird, wurde bereits in Abbildung 5.8 gegeben. Die entsprechenden sieben Phasen werden nun einmal durchlaufen, um das vorgestellte Beispielsystem zu entwickeln. Zur Entwicklung wurde das AgentTool 2.0 (Beta)[2] benutzt. Es unterstützt die Erstellung der von MaSE verwendeten Diagramme.

In der ersten Phase (**Capturing Goals**) werden die Ziele des zu entwickelnden Systems strukturiert. Das System ist in diesem Beispiel sehr übersichtlich. Das eigentliche Ziel ist es einzelne Sätze zu übersetzen. Erreicht wird dies durch die Übersetzung der einzelnen Wörter. Es ergibt sich eine Ziel–Hierarchie wie in Abbildung 6.2 gezeigt. In **Applying Use Cases** werden aus den Anforderungen an das System einzelne Anwendungsfälle gewonnen. Die schriftliche Beschreibung eines Anwendungsfalles wird durch mindestens ein Sequenzdiagramm verdeutlicht. Im Mittelpunkt der Entwicklung steht die Kommunikation zwischen den zukünftigen Agenten. Daher soll früh eine Vorstellung davon entwickelt werden, welche Rollen miteinander kommunizieren. Der Ablauf einer Übersetzung wurde bereits in Abbildung 6.1 gezeigt. Abbildung 6.3 zeigt ein entsprechendes MaSE–Sequenzdiagramm. In **Refining Roles** werden nun aus den erkannten Zielen Rollen identifiziert. Im Zweifelsfall wird für jedes Ziel des Systems eine eigene Rolle zu dessen Erfüllung modelliert. Außerdem werden den Rollen *Tasks* zugeordnet. Sie beschreiben das Verhalten der Rollen. Die Rollen werden im *role model diagram* (Abbildung 6.4)

[2] http://www.cis.ksu.edu/s̃deloach/ai/projects/agentTool/agentool.htm

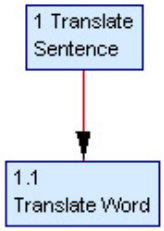


Abbildung 6.2: MaSE - Goal Hierarchy Diagram

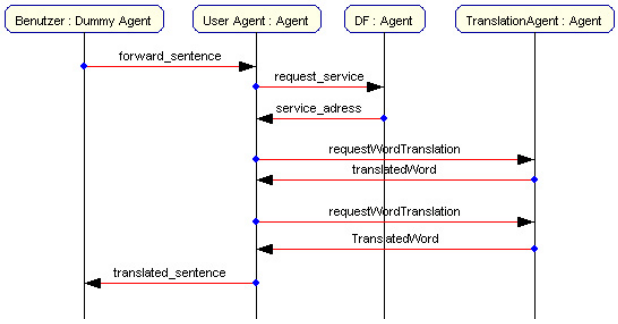


Abbildung 6.3: MaSE - Sequenzdiagramm der Übersetzung

dargestellt. Die Nummern in den Rechtecken referenzieren die Ziele aus dem Goal Diagram. Die Rollen *DF* und *Benutzer* referenzieren keine Ziele. DF wird vom System gestellt, die interne Arbeitsweise interessiert im Weiteren nicht, der Benutzer wird nur modelliert, um die Kommunikation beschreiben zu können. Die Pfeile zwischen den einzelnen Tasks (die Ovale, die den Rollen zugeordnet sind) repräsentieren Protokolle zwischen den Rollen. Sie zeigen vom Initiator der Kommunikation zum Antwortenden. Die einzelnen Tasks werden in UML–Zustandsdiagrammen beschrie-

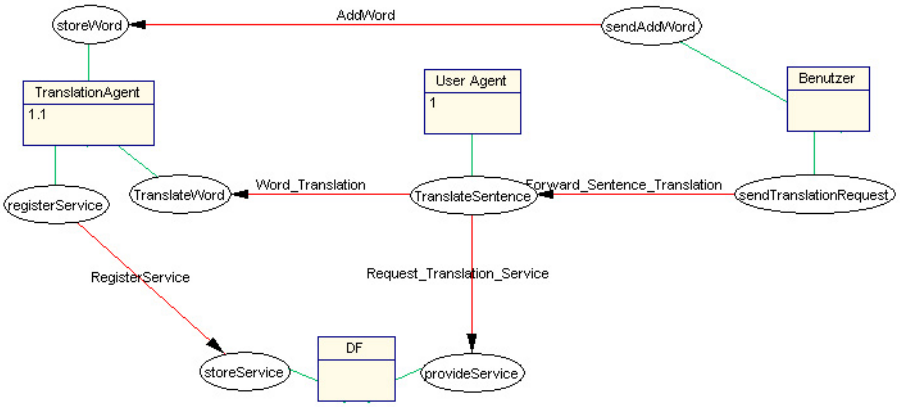


Abbildung 6.4: MaSE - Das Rollenmodell

ben. Abbildung 6.5 zeigt ein solches Diagramm für das Task *TranslateSentence*. Der Ablauf ist denkbar einfach. Der Empfang einer Nachricht (*forwardSentence()*) startet den Task. Es werden die Wörter des empfangenen Satzes extrahiert (Aktion des Übergangs zum Zustand *translateSentence*) und so lange die Übersetzungen der einzelnen Wörter angefragt, bis alle Wörter übersetzt sind. Dann wird mit einer Antwort–Nachricht, der Task beendet.

Die gefundenen Rollen werden in der Phase **Creating Agent Classes**, zu Klassen

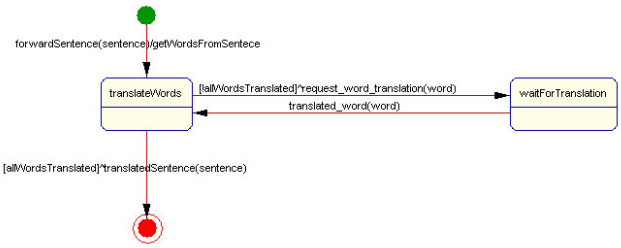


Abbildung 6.5: MaSE - Task Diagramm für TranslateSentence

von Agenten zusammengefasst. Die Klassen *DummyAgent* und DF werden von der Jade Plattform bereitgestellt. Die anderen beiden Rollen werden jeweils durch einen eigenen Agenten in Jadex realisiert. Das *Agent Class Diagram* (Abbildung 6.6) zeigt die Klassen und die Rollen, die sie erfüllen. Verbunden sind die Agenten durch die Kommunikationen zwischen ihnen (in MaSE *conversations* genannt).

Constructing Conversations spezifiziert diese Conversations der vorigen Phase.

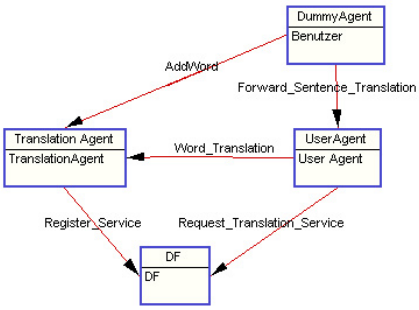


Abbildung 6.6: MaSE - Agent Class Diagram

Sie definieren die Koordinationsprotokolle zwischen zwei Agenten und werden in *Conversation Class Diagrams* (UML–Zustandsdiagramme) beschrieben. Es existieren immer zwei dieser Diagramme, eins für den Initiator, eins für den Antwortenden. Abbildung 6.7 zeigt beide Diagramme zur Beschreibung der Conversation *RequestWordTranslation*. Die initiierende Seite sendet durch *request_word_translation(word)* eine Anfrage an den Translation Agent, ein Wort zu übersetzen. Danach wird auf die Antwort gewartet. Die eckigen Klammern beschreiben Bedingungen für den Übergang in einen anderen Zustand. Im unteren Teil der Zustände werden auszuführende Aktionen genannt. Es wird auf die verschiedenen möglichen Antworten reagiert. Anschließend wird der Endzustand erreicht. Die antwortende Seite bearbeitet die Anfrage. Die Aktion *queryWordFromBeliefBase* kann entweder ein Wort finden oder nicht. Entsprechend den angegebenen Bedingungen wird eine Antwort (symbolisiert durch ein $\wedge$) zurückgesandt.

In **Assembling Agent Classes** wird der interne Aufbau eines Agenten beschrie-

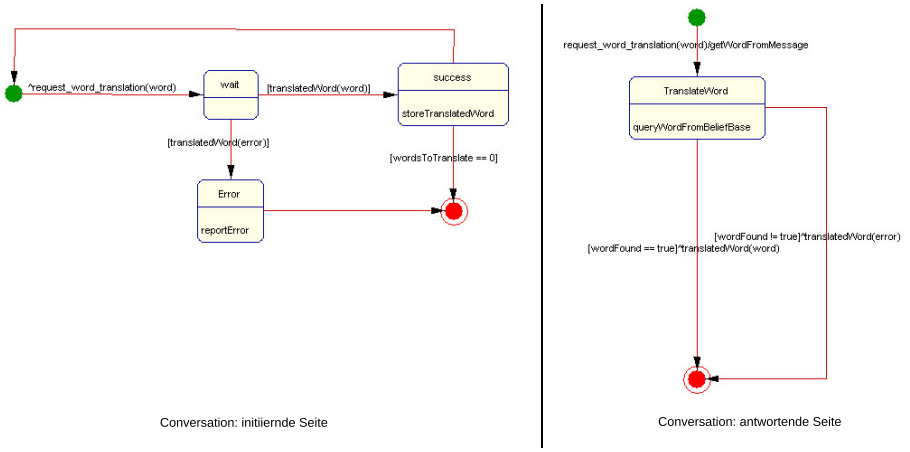


Abbildung 6.7: MaSE - Beide Diagramme zur Beschreibung der Übersetzung eines Wortes

ben. Diese Phase ist eng verknüpft mit der Vorigen, da die Art, wie die Agenten

die ausgetauschten Nachrichten verarbeiten, in großem Maße vom internen Aufbau der Agenten abhängt. MaSE will plattformunabhängig sein. Zur Repräsentation des Aufbaus werden abgewandelte Klassendiagramme vorgeschlagen. Sie unterscheiden zwischen Assoziationen zu internen Klassen und anderen Agenten.
Im **System Design** werden (UML–)Implementierungsdiagramme verwendet, um die Typen, Anzahl und Position von Agenten im System zu zeigen. Hier wird die Verteilung der Agenten auf die verfügbaren Agentenplattformen gezeigt. in Abbildung 6.8 befinden sich alle Agenten auf der selben Plattform.

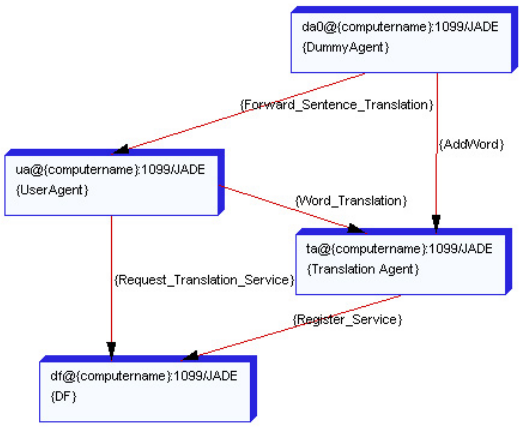


Abbildung 6.8: MaSE - Deployment Diagram aller Agenten

Bemerkungen zur Entwicklung

Diese Methode legt besonderen Wert auf die Modellierung der Kommunikation zwischen den Agenten. Die Agenten werden als Zustandsmaschinen verstanden und die Agenten dementsprechend durch Zustandsdiagramme beschrieben. Die Unterstützung der BDI-Konzepte wird für die Phase Assembling Agent Classes vorgeschlagen, sie werden nicht während der gesamten Entwicklung benutzt. Dies schmälert den Wert der Methode. Als *Modellierungssprache* wird in der Designphase auf Notationen der UML zurückgegriffen. Dies erhöht die Benutzbarkeit. Die Notationen sind Entwicklern bekannt und auch per Hand einfach zu zeichnen. Die Notation des Nachrichtenversands in den Zustandsdiagrammen ist gewöhnungsbedürftig, aber eingängig. Die eigens entwickelte Notationen zur Beschreibung der beteiligten Rollen eines System ist den Klassendiagrammen der UML nicht unähnlich und bestärkt so den Eindruck, dass UML gewohnte Entwickler sich kaum umstellen müssen. Die Modelle sind eindeutig definiert. Verfeinerung und Modularisierung sind mit den selben Mitteln der UML gegeben. Der Aufbau der Modelle ist nachvollziehbar und führt den Entwickler von einer groben Vision zum Design eines Agentensystems. Insbesondere das verwendete Werkzeug (agentTool) unterstützt die Durchgängigkeit der Modelle, d. h. Designartefakte können in verschiedenen Sichten betrachtet werden.
Die *pragmatischen Gesichtspunkte* werden bestimmt durch den guten Eindruck, den das CASE–Tool *agentTool* hinterließ. Die Modelle können hiermit sehr leicht erstellt

und geändert werden. Der Eindruck des Werkzeugs wird präzisiert durch den ausgefüllten ISONORM–Fragebogen, der in Anhang D zu finden ist. Das Programm ist aufgabenangemessen, ist gut steuerbar und verhält sich erwartungskonform. Es ist kaum selbstbeschreibend, schließlich richtet es sich an Entwickler, die mit der MaSE–Methode vertraut sind. Es verhält sich fehlertolerant, aber zur Behebung von Fehlern werden kaum Hinweise gegeben. Die Benutzungsoberfläche ist nicht individualisierbar, aber einleuchtend aufgebaut, was zur Lernförderlichkeit beiträgt. Als Dokumentation zur Methode sind lediglich Beiträge zu Konferenzen vorhanden. Nach der Unitersuchung von Damm und Winikoff (2003) wurde diese Methode am häufigsten in universitären Projekten eigesetzt, eine kommerzielle Benutzung fand noch nicht statt.

6.3 Entwicklung nach Tropos

Die Entwicklung nach Tropos gliedert sich in vier Phasen. Die Early Requirements, die Late Requirements, das Architectural Design und das Detailed Design (siehe Kapitel 5.3.4).

In den **Early Requirements** wird nicht das zu entwickelnde System, sondern die Umgebung, in der es zukünftig arbeiten soll, modelliert. Da ein System für nur eine Art Benutzer (Der die GUI des *Dummy Agent* benutzt) entwickelt wird, kann in dieser Phase nur ein Akteur identifiziert werden (Ansonsten würden die Beziehungen zwischen den Akteuren in Actor–Diagramm dargestellt werden). Dieser hat zwei Ziele. Das Hardgoal, Sätze übersetzen zu lassen, und das Softgoal, dass die Übersetzung so effizient wie möglich verläuft. Abbildung 6.9 zeigt ein entsprechendes Goal–Diagramm. Der eigentliche Zweck dieser Phase ist es, die Ziele und Interessen aller beteiligten Parteien an dem System zu modellieren. Würde unser System also in einem größeren Kontext eingesetzt werden (von einer Organisation zur Verfügung gestellt werden, Teil einer Applikation sein, etc.), wäre diese Phase sinnvoll. In diesem kleinen Rahmen macht sie wenig Sinn.
Die anschließenden **Late Requirements** betrachten das zukünftige System im Zu-

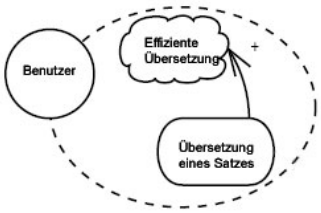


Abbildung 6.9: Tropos - Die Ziele des Benutzers

sammenspiel mit der vorher beschriebenen Umgebung. Das zu entwickelnde System wird als Aktor mit Abhängigkeiten zu den anderen Aktoren der Umgebung beschrieben. Abbildung 6.10 zeigt so ein Goal–Diagram. Das eingeführte System (als *System* eingezeichnet) soll dem Benutzer beide Ziele erfüllen. Das Softgoal der effizienten Übersetzung und die eigentliche Übersetzung eines einzelnen Satzes. Bereits in die-

ser frühen Phase bietet Tropos die Möglichkeit, die ersten (weil offensichtlichen) Pläne zur Erfüllung der Hardgoals zuzuordnen. So erfüllt der Plan *EnglishGerman-TranslateWordPlan* das Ziel, Wörter zu übersetzen.

Im **Architectural Design** wird nun in drei Schritten die Architektur des Systems

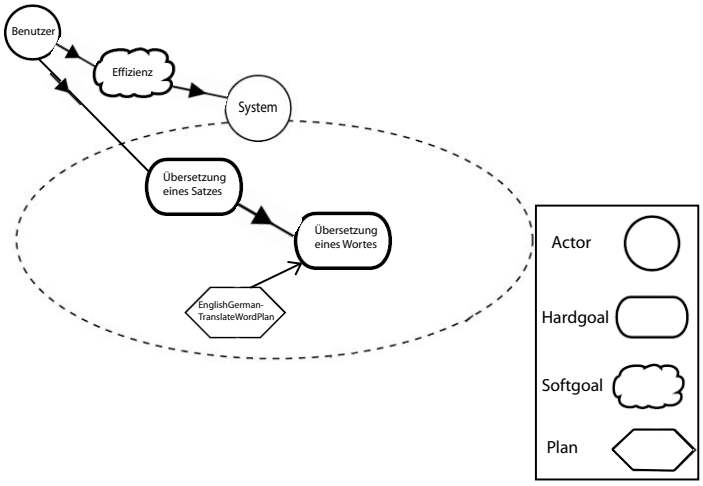


Abbildung 6.10: Tropos - Das zu entwickelnde System

entwickelt. Im ersten Schritt werden dem System weitere Akteure hinzugefügt. Jeder unserer Agenten soll ein Ziel erfüllen, also werden zwei Agent eingeführt. Der *User Agent* ist dafür zuständig, ganze Sätze zu übersetzen. Dazu bedient er sich des *Translation Agent*, der die Übersetzung der einzelnen Wörter übernimmt. Abbildung 6.11 zeigt diese Hierarchie.

Das Ergebnis dieses ersten Schrittes ist das Extended–Actor–Diagram (Abbildung 6.12). Es zeigt die neu eingeführten Akteure und ihre Abhängigkeiten mit anderen Akteuren. Die Kästen stellen die Objekte dar, die zwischen den Agenten ausgetauscht werden. Die Pfeile geben die Richtung der Benutzung der Objekte an. Im zweiten Schritt werden die benötigten Capabilities identifiziert. Sie sind aus den Abhängigkeiten in dem eben gesehenen Diagramm ableitbar. So hängt der Translation Agent davon ab, Wörter zur Übersetzung und zur Erweiterung seiner Wissensbasis zu erhalten. Tropos schlägt vor, alle Abhängigkeiten als Capabilities zu modellieren und diese dann gegebenenfalls zusammenzufassen. Da die selben Daten (ein Wort) verarbeitet werden und auf die selbe Wissensbasis zugegriffen wird, bietet sich die Zusammenfassung zur *Translation Capability* an.

Der dritte und letzte Schritt identifiziert die verschiedenen Arten (Klassen) von Agenten und ordnet ihnen verschiedene der gefundenen Capabilities zu. Das Ergebnis wird in Tabelle 6.1 dargestellt.

Das **Detailed Design** spezifiziert nun die Agenten. In drei Arten von Diagrammen werden die Capabilities, die Pläne und die Interaktionen zwischen den Agenten abgebildet. Die Capabilities und Pläne werden jeweils durch UML–Aktivitätsdiagramme beschrieben, die Interaktionen durch AUML–Interaktionsdiagramme. Interessanterweise werden auch die Capabilities als Aktivitäten abgebildet (und nicht nur als einfache Sammlungen von Plänen, Beliefs, Events und Regeln für deren Sichtbarkeit nach außen, wie in von Busetta et al. (1999) beschrieben). Die Aktivitäten in den

Abbildung 6.11: Tropos - Zwei Akteure (Kreis) für zwei Ziele

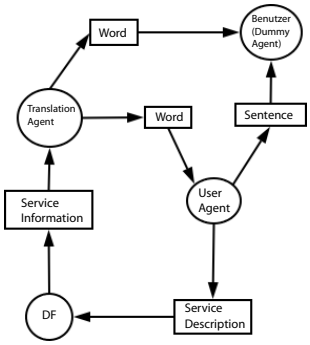


Abbildung 6.12: Tropos - Extended Actor Diagram

Agent	Capabilities
User Agent	DFCapability
Translation Agent	DFCapability, TranslationCapability

Tabelle 6.1: Tropos - Die Capabilities der Agenten

Aktivitätsdiagrammen sind die aufrufbaren Pläne. Hierdurch wird der Kontrollfluß durch die Agenten leicht nachvollziehbar. Abbildung 6.13 zeigt die TranslationCapability. Zwei Events können den Ablauf dieser Capability starten (Die Abkürzung *EE* steht für external Event). Hier sind diese Events der Empfang einer Anfrage. Entsprechend den Anfragen wird entweder der Plan zur Übersetzung eines Wortes, oder zum Einfügen eines Wortes in die BeliefBase ausgeführt. Die einzelnen Zugriffe auf die Beliefbase können in den Plänen beschrieben werden, aber in diesem Diagramm wird sie mit eingetragen, um anzuzeigen, dass auch sie Teil der Capability ist. Diese Capability enthält auch den EnglishGermanTranlsationPlan. Der

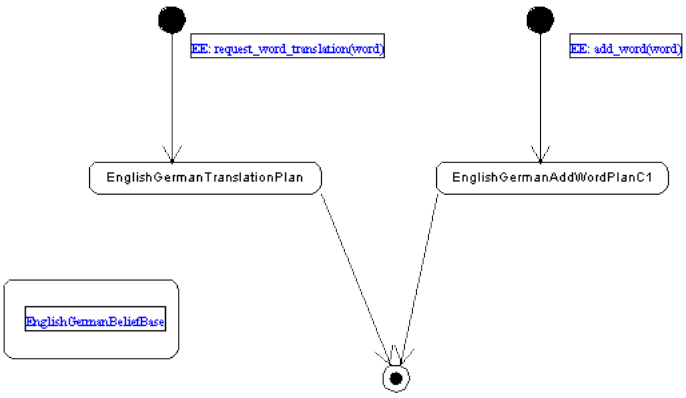


Abbildung 6.13: Tropos - Capability Diagram: Die TranslationCapability

Ablauf dieses Planes wird in Abbildung 6.14 dargestellt. Das zu übersetzende Wort wird aus der empfangenen Nachricht extrahiert, in der Beliefbase gesucht und dem Suchergebnis entsprechend eine Antwort gesendet. Die Interaktionen zwischen den einzelnen Agenten werden nach Tropos in Interaktionsdiagrammen der AUML dargestellt. Abbildung 6.15 zeigt den Austausch der Nachrichten um eine Folge von zwei Wörtern zu übersetzen. Durch die erste Nachricht wird beim User Agent angefragt, den übertragenen Satz zu übersetzen. Wichtig ist der Unterschied in der Semantik der Nachrichten. Es handelt sich nicht um Aufrufe – das wäre ein objektorientierter Mechanismus – sondern um Anfragen. Die Agenten können sich immer dazu entscheiden einer Anfrage nicht nachzukommen. Um die Übersetzung auszuführen, wird dann vom DF die Adresse des Service zur Übersetzung von Wörtern erfragt, um dann Anfragen zur Übersetzung der einzelnen Wörter abzusetzen. Wurden alle Wörter übersetzt, wird das Ergebnis an den Aufrufer übermittelt.

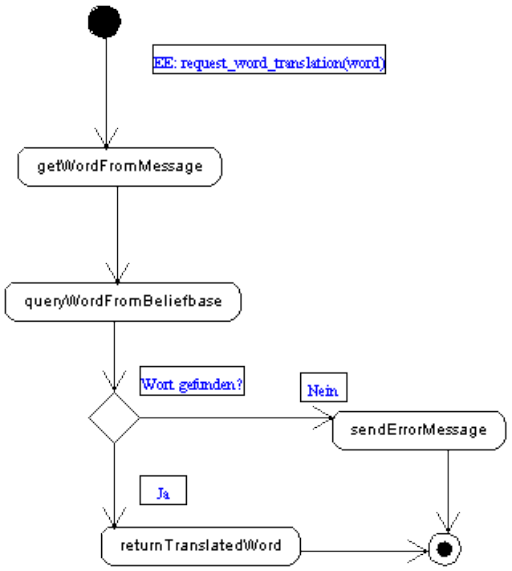


Abbildung 6.14: Tropos - Plan Diagram: EnglishGermanTranslateWordPlan

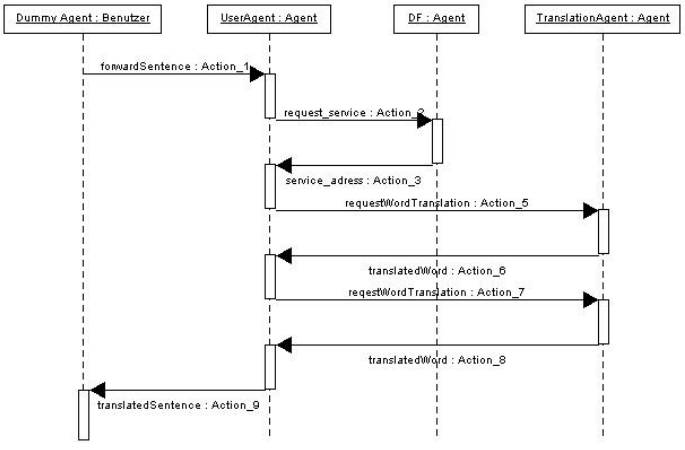


Abbildung 6.15: Tropos - Interaktionsdiagram

Bemerkungen zur Entwicklung

Die frühe Phase der Anforderungsnanalyse war unnötig für das hier entwickelte System. Sie versucht eher die Situation zu modellieren, in der das System benötigt wird. Die Modellierungssprache zeichnet sich dadurch aus, dass verhältnismäßig wenig Modelle verwendet werden. Nachdem in den Goal– und Actordiagrammen das System bechrieben wurde, werden ihm noch die einzelnen Akteure im Extended–Actor–Diagram beigefügt. Dann wird schon die Implementation der Agenten durch Aktivitätsdiagramme und Interaktionsdiagramme spezifiziert. Dies zeigt, wie die Methode sich auf die Modellierung von Organisationen konzentriert, die spätere Modellierung der Agenten ist nicht so detailliert, wie in den anderen Methoden. Bemerkenswert ist hierbei, dass auch die Capabilities durch Aktivitätsdiagramme dargestellt werden. Im Gegensatz zu einer einfachen Beschreibung der enthaltenen Elemente, ist so der Kontrollfluss durch die Agenten und deren Capabilities nachzuvollziehen. Die im detaillierten Design verwendeten Notationen der UML sind leicht benutzbar. Allerdings werden Akteure nicht voneinander unterschieden. Ein Akteursymbol kann eine physisch vorhandene Person, einen Softwareagenten oder eine Rolle repräsentieren. Dies führt in großen Modellen zu Unübersichtlichkeit. Da die Methode sich auf die Modellierung von Organisationen konzentriert, ist die Ausdrucksstärke bezüglich der Struktur und des Aufbaus eines Agentensystems nicht so stark wie in den anderen Methoden. Es existieren weniger Sichten auf das System. Dafür ist die schrittweise Verfeinerung und der Aufbau der Modelle aufeinander sehr übersichtlich. Diese Vorgehensweise und die Modelle leiten den Entwickler darin an, das Softwaresystem als Organisation zu begreifen. Diese spezielle Sicht leitet insbesondere unerfahrene Entwickler in diesem neuen Paradigma an. Die Definition der Modellierungssprache eindeutig. Ungewöhnlich ist hierbei, dass die Akteure sowohl Benutzer, Rollen als auch konkrete Agenten beschreiben. Im Detailed Design werden nur UML und AUML Diagramme verwendet. Dies ist praktisch, da hierfür eine Vielzahl von Werkzeugen besteht. Zur Modellierung in den frühen Arbeitsschritten existieren Werkzeuge die die i*–Notation unterstützen. Da es kein Werkzeug gibt, um den gesamten Entwicklungsprozess zu unterstützen, sind die Entwickler in der Wahl ihrer Werkzeuge frei. Hauptsächlich werden sie UML–CASE–Werkzeuge benutzen, um das Design der Agenten zu spezifizieren. Daher entfällt eine Bewertung mittels des ISONORM–Fragebogens. Zur Dokumentation der Benutzung der Methode sind Beiträge zu Konferenzen und Zeitschriften vorhanden. Von den drei betrachteten Methoden wurde Tropos bisher am wenigsten in (universitären) Projekten eingesetzt (nach Dam und Winikoff, 2003).

6.4 Entwicklung nach Prometheus

In Kapitel 5.3.5 wurde der Entwicklungsprozess nach Prometheus vorgestellt. Er beginnt mit der Phase der **System Specification**. Hier werden sowohl die Actions und Percepts, als auch die Goals und Functionalities des zukünftigen Systems festgestellt. Da es sich bei dem zu entwickelnden System um ein rein kommunikatives Multiagentensystem handelt, existieren keine Actions. Die Umwelt des Systems wird nicht beeinflusst. Als Percepts kommen nur die Veränderungen des Benutzers an der GUI des *Dummy–Agenten* in Frage. Das einzige Ziel des Systems ist die Übersetzung von Sätzen. Dies wird durch Teilziele, die Übersetzung der einzelnen Wörter,

erreicht. Die Funktionalities beschreiben die Fähigkeiten, die nötig sind, um die Ziele zu erreichen. So wird zur Übersetzung eines Wortes offensichtlich die Funktionalität benötigt, die Wörter in der verwendeten Beliefbase zu suchen. In hierarchischen Strukturen werden die Goals und die Funktionalities, die zu ihrer Erfüllung beitragen, geordnet. In dieser frühen Phase benutzt Prometheus sog. Scenarios, um die Arbeitsweise des Systems zu verdeutlichen. Sie zeigen eine exemplarische Abfolge von Arbeitsschritten, die so im System stattfinden könnte. Der geplante Arbeitsablauf des Systems aus Abbildung 6.1 stellt ein solches Scenario dar. In Prometheus würde dieses allerdings nicht durch ein Diagramm, sondern durch einen Scenario Descriptor in schriftlicher Form dargestellt werden.
Nachdem ein Überblick über die Anforderungen an das System gewonnen wurde, wird in der folgenden Phase, dem **Architectural Design**, Aufbau und Eigenschaften des gesamten Systems beschrieben. Die folgenden Diagramme wurden mit dem *Prometheus Design Tool* (siehe Abschnitt 5.3.5) erstellt. Das System–Overview–Diagramm (Abbildung 6.16) zeigt den Aufbau des Systems. Die Agenten werden durch die Kästen mit den enthaltenen Strichmännchen dargestellt. Das System wurde in die vier bekannten Agenten aufgeteilt. Veränderungen in der GUI des Dummy Agent werden als Percept dargestellt. Die Pfeilsymbole, die die Agenten miteinan-

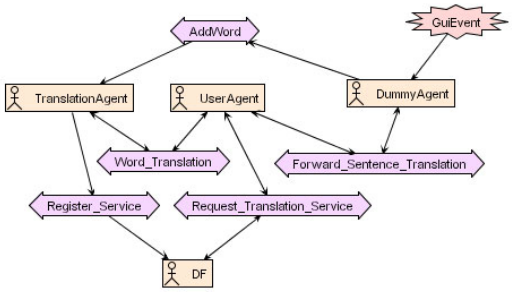


Abbildung 6.16: Prometheus - System Overview Diagramm

der verbinden, repräsentieren Protokolle zwischen den Agenten. Diese Protokolle enthalten die Abfolge der Nachrichten, die zwischen den Agenten ausgetauscht werden (siehe Abbildung 6.17). Das zeitliche Verhalten der Nachrichten wird in

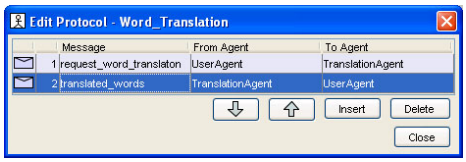


Abbildung 6.17: Prometheus - Beschreibung eines Protokolles

Interaktionsdiagrammen der AUML ausgedrückt (siehe Abbildung 6.15 - es zeigt die Abfolge der Nachrichten, um einen Satz aus zwei Wörtern zu übersetzen). Das Agent–Aquaintance–Diagramm (Abbildung 6.18) zeigt die Agenten und die Kommunikationsverbindungen unter ihnen. Es existiert nur eine Beliefbase im System, die

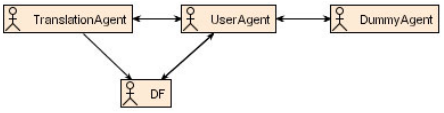


Abbildung 6.18: Prometheus - Agent Aquaintance Diagram

ausschließlich vom *Translation Agent* benutzt wird. Im Data–Coupling–Diagramm werden die Datenbestände und die Functionalities, die auf sie zugreifen (lesend und/oder schreibend), dargestellt. In diesem Fall ist das Diagramm (Abbildung 6.19) denkbar einfach. Eine Functionality greift lesend auf die Daten (*EnglishGermanbeliefBase*) zu (*lookUpWord*), und eine Andere greift schreibend zu (*AddWordToBeliefBase*).

Für den Übergang in die nächste Phase, das **Detailed Design**, beschreiben die

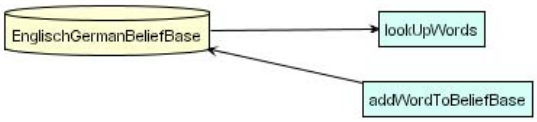


Abbildung 6.19: Prometheus - Data Coupling Diagramm

Agent–Descriptors die Eigenschaften der Arten von Agenten, die für das System gefunden wurden. Diese Phase spezifiziert nun den Aufbau der Agenten. Abbildung 6.20 zeigt ein Agent–Overview–Diagramm für den Translation Agent. Hier werden die Capabilities des Agenten und die eingehenden und ausgehenden Nachrichten gezeigt. Die Capabilities wiederum werden durch Deskriptoren spezifiziert. Beide Agenten enthalten die Capability *DFCapability*. Die Tabelle 6.2 zeigt die beiden Agent Descriptors. Neben weiteren Details werden hier die empfangenen und gesendeten Nachrichten, die verwendeten Protokolle, die Ziele, Pläne und Capabilities der Agenten aufgelistet. Eine komplette Liste der einzelnen Attribute und ihrer Bedeutung wurde von Padgham und Winikoff (2002) gegeben. Abbildung 6.21 zeigt das entsprechende Capability–Overview-Diagramm für die *Translation Capability*, die durch die den Agent Descriptor (6.3) beschrieben wird.

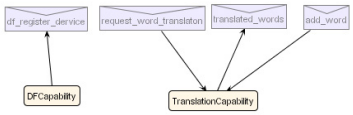


Abbildung 6.20: Prometheus - Der Translation Agent

Auch die Pläne, Beliefbases, Events und und in der Beliefbase enthaltene Daten können durch entsprechende Deskriptoren beschrieben werden. Diese werden hier nicht mehr vorgestellt, da der Leser nach den oben beschriebenen Deskriptoren einen Eindruck erhalten hat, wie detailliert die einzelnen Elemente des Systems

Agent Descriptor:	User Agent	Translation Agent
Description	Nimmt Anfragen zur Übersetzung ganzer Sätze entgegen und gibt die Übersetzung zurück	Nimmt Anfragen zur Übersetzung einzelner Wörter entgegen, übersetzt diese selber und gibt die Übersetzungen zurück
Cardinality Minimum	1	1
Cardinality Maximum	1	1
Creation		
Destruction		
Initialisation		
Demise		
Incomming Messages	translated_words, forward_sentence,	request_word_translaton, add_word service_adress
Outgoing Messages	request_word_translaton, translated_sentence	translated_words, df_register_service
Internal Messages		
Percepts		
Actions		
Read Data		EnglishGermanBeliefbase
Written Data		
Internal Data		
Goals	Translate_Sentence	Translate_Word
Functionalities	DelegateTranslationOfWords	lookUpWord
Protokolls	Word_Translation, Forward_Sentence_Translation, Request_Translation_Service	Word_Translation, Register_Service, AddWord
Included Plans	EnglishGermanTranslateSentencePlan	EnglishGermanTranslateWordPlan
Included Capabilities	DFCapability	DFCapability, TranslationCapability

Tabelle 6.2: Prometheus - Agent Descriptors

Capability Descriptor:	TranslationCapability
Description	Ermöglicht die Übersetzung einzelner Wörter, sowie das Einfügen neuer Wörter in die Beliefbase
Cardinality	1
Lifetime	
Goals	1
Included Plans	EnglishGermanAddWordPlan, EnglishGermanTranslationPlan
Included Capabilities	
Internal Data	EnglischGermanBeliefBase
Read Data	egwords
Written Data	egwords
Actions	
Percepts	
Incomming Messages	request_word_translaton, add_word
Outgoing Messages	translated_words
Internal Messages	

Tabelle 6.3: Prometheus - Die Translation Capability

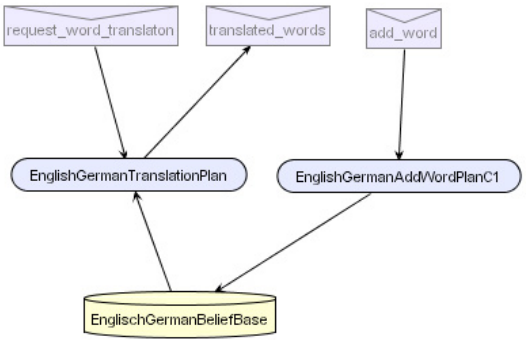


Abbildung 6.21: Prometheus - Übersicht über die TranslationCapabilty

beschrieben werden können.

Bemerkungen zur Entwicklung

Insbesondere durch die detaillierten Deskriptoren wird das zu implementierende System sehr genau beschrieben. Zentrale Elemente von Prometheus sind die Übersichtsdiagramme über das System und die einzelnen Agenten. Diese ungewöhnliche und ausnahmsweise von der UML nicht beeinflusste Notation ist eindeutig definiert. Die Modelle sind ohne CASE–Werkzeug schwer zu entwickeln, für die Nutzung der Methode ist unentbehrlich, die Deskriptoren über die grafischen Modelle navigieren zu können. Viele implementationsbestimmende Details werden in ihnen beschrieben. Dies macht die Modelle ausdrucksstärker als in den anderen Methoden. Verfeinerung, Modularisierung, Durchgängigkeit und aufeinander aufbauende Modelle sind gegeben. Unter den pragmatische Gesichtspunkten besticht die Werkzeugunterstützung für die Methode (Die verfügbaren Werkzeuge wurden bereits in Abschnitt 5.3.5 beschrieben). Der Eindruck, den die Benutzung des PDT vermittelte, wird in dem ausgefüllten ISONORM-Fragebogen in Anhang D genauer beschrieben. Das Programm ist angemessen für die Erstellung der Diagramme, allerdings wenig selbstbeschreibend. Die Steuerbarkeit ist insgesamt gut, Benutzer leiden aber unter kleineren Fehlern der Darstellung der Elemente. Ansonsten verhält sich das Programm erwartungskonform. Die Oberfläche ist nicht individualisierbar, aber einfach strukturiert, was die Erlernbarkeit fördert. Während der Benutzung wurden kleinere Fehler in der Darstellung gefunden, auch die interne Konsitenzüberprüfung lieferte zuweilen unsinnige Ergebnisse. Leider gaben die Entwickler, auf Nachfrage, hierzu keinen Kommentar. Neben Konferenzbeiträgen ist auch ein Tutorial zur Benutzung von Prometheus (Padgham, 2002) verfügbar. Ähnlich MaSE wurde diese Methode in einigen universitären Projekten eingesetzt (nach Dam und Winikoff, 2002).

6.5 Abschließende Bewertung

Die *Modellierungssprache* basiert in allen Methoden auf eigens entwickelten Notationen. In den Arbeitsschritten des Designs verwenden MaSE und Tropos (abgewandelte) Diagramme der UML. Dies erleichtert die Benutzung, da diese Diagramme weit verbreitet sind. In den frühen Arbeitsschritten verwendet Tropos die ungewohnte Notation von i* (siehe Kapitel 5.3.4). In dieser Notation wird nicht zwischen Agenten und Rollen, die die Agenten einnehmen können, unterschieden. Dies macht größere Modelle unübersichtlich. Prometheus verwendet eine eigens entwickelte Notation, die nur durch eigene Werkzeuge unterstützt wird.
Die Ausdrucksstärke der Modelle scheint vergleichbar. Sie alle bemühen sich, sowohl die Systeme, als auch die Architekturen der einzelnen Agenten zu veranschaulichen. Bei MaSE steht hierbei die Kommunikation innerhalb des Systems im Vordergrund. Auch der Kontrollfluß durch die Agenten wird anschaulich modelliert. Tropos konzentriert sich in den frühen Arbeitsschritten auf die Identifikation der Ziele und Abhängigkeiten der Agenten. Während des Designs wird der Kontrollflusses im System modelliert. Prometheus beschreibt detaillierter als die anderen Methoden den letztendlichen Aufbau der Agenten (vor allem durch die Deskriptoren). Die weiteren wünschenswerten Eigenschaften der Modellierungssprache (Verfeinerung, logischer Aufbau der Modelle aufeinander und Durchgängigkeit der Artefakte) sind gegeben. Die Unterstützung durch Werkzeuge bestimmt die *pragmatischen Gesichtspunk-*

te. MaSE wird durch das AgentTool [3] (siehe Abschnitt 5.3.2) unterstützt. Die Prometheus–Diagramme wurden mit dem PDT [4] (siehe Abschnitt 5.3.5) erstellt. Es handelt sich um einfache grafische Editoren. Neben der Erstellung der Diagramme wird die Konsistenz der Modelle überprüft. Für Tropos existieren Werkzeuge, die die i*–Notation[5] (siehe Abschnitt 5.3.4) unterstützen. Zusätzlich werden UML–Diagramme verwendet. Diese Werkzeuge sind sehr einfach gehalten. Sie sind ähnlich leicht einsetzbar, mit ihnen wurden die Diagramme erstellt. Die Dokumentation der verwendeten Werkzeuge ist äußerst kurz. Da es sich aber lediglich um Zeichenwerkzeuge für die Diagramme handelt, wird hierdurch nur die Einarbeitungszeit verlängert. Die Dokumentation der Methoden an sich besteht weitestgehend aus Beiträgen zu Konferenzen oder Zeitschriften. Prometheus wurde zusätzlich in einem Workshop gelehrt. Die entsprechenden Unterlagen sind online verfügbar[6]. Zu allen Methoden ist die Dokumentation ausführlich und führt, anhand von Beispielen, in deren Verwendung ein. Nach der Umfrage von Dam und Winikoff (2003) (in der auch die Autoren befragt wurden) wurde keine der Methoden in einem kommerziellen Projekt verwendet. Die Verwendung in universitären Projekten wurde mittels eines (Multiple–Choice–) Fragebogens ermittelt. Nach MaSE wurden hiernach mehr als 21, nach Prometheus sechs bis 20 und nach Tropos eine bis fünf Anwendungen entwickelt. Die Entwickler der Methoden wurden weiterhin nach deren Zielgruppen befragt. Nach deren Aussage, richten sich MaSE und Prometheus an Programmierer (sowohl Studenten als auch in der Industrie), für Tropos wurde "Experten" angegeben.

Fazit

Alle drei Methoden, die in die 'engere Wahl'[7] kamen (MaSE, Tropos, Prometheus), sind geeignet, als Grundlage zur Entwicklung von Agentensystemen mit Jadex zu dienen. Allerdings unterstütz MaSE die BDI-Konzepte nicht in dem Umfang wie die anderen beiden Methoden.[8] Da aber gerade diese für Jadex im Vordergrund stehen, ist dies ein großer Nachteil. Prometheus besticht durch die detaillierten Beschreibungen der einzelnen Komponenten eines Agenten und ein eigenes frei verfügbares Werkzeug.[9]
Zur Bewertung von agentenorientierten Methoden schlagen Yu und Cysneiros (2002) die Anwendung eines sog. *challenge exemplar* vor. Sie bemängeln, dass nicht alle Beispiele zur Veranschaulichung und Bewertung von Methoden gleich gut geeignet sind. Deshalb schlagen sie eine umfassende Problemstellung vor, um die zu vergleichenden Methoden in einem gemeinsamen Kontext mit einem gut verstandenen Problem, von ausreichender Komplexität, zu erproben. Als "Herausforderungsproblem" wird ein umfangreiches, medizinisches Informationssystem vorgeschlagen. Es ist leicht einzusehen, dass die Modellierung eines komplexen Problems, deutlicher die Stärken und Schwächen von Modellierungssprachen, Analyse- und Designprozessen zeigt. In der Betrachtung des hier verwendeten Fallbeispieles zeigte sich die Phase der Early Requirements als unnötig. In dem Rahmen dieser Arbeit konnte ein solches Callenge Exemplar nicht berücksichtigt werden. Der Zeitaufwand für eine

[3] Version: 2.0 (Beta)
[4] Version: 1.1
[5] Übersicht unter: http://www.troposproject.org/
[6] http://www.cs.rmit.edu.au/agents/
[7] Beschreibung der Vorauswahl: siehe Abschnitt 5.4
[8] beschrieben in Abschnitt 5.3.6
[9] Methode vorgestellt in den Abschnitten 5.3.5 und 5.3.6

solch ausführliche Modellierung nach verschiedenen Methoden würde eine eigene Arbeit rechtfertigen. Hoffentlich greifen zukünftige Arbeiten diese Anregung auf. In einer derartigen Gegenüberstellung sind insbesondere interessante Ergebnisse für die Möglichkeiten der Modellierung von Zielen, die Early Requirements konzentrieren sich hierauf, zu erwarten.

Hier wird die Verwendung von Prometheus zur Unterstützung der Entwicklung mittels Jadex vorgeschlagen. Die benutzten Konzepte stimmen weitestgehend überein.[10] Das Vorgehensmodell ist ähnlich umfangreich wie in Tropos, die Modellierungssprache leicht umfangreicher. Ein Werkzeug ist frei erhältlich, und die verfügbare Dokumentation ist ausführlich und leicht verständlich.

[10] siehe Abschnitt 5.3.6

Kapitel 7

Generierung von Agent Definition Files aus Design–Artefakten

Zur Unterstützung der Anwendungsentwicklung wurde ein Prototyp programmiert, der die Erstellung der XML–Dateien unterstützen soll, die die einzelnen Agenten der Laufzeitumgebung des Jadex Systems bekannt machen. Ihre Funktion wurde bereits in Abschnitt 2.6.1 vorgestellt, im folgenden Abschnitt 7.1.1 wird ihr Aufbau detailliert beschrieben. Die verwendeten Werkzeuge, der Aufbau und die Funktionsweise der Software werden vorgestellt, bevor dessen Anwendbarkeit untersucht wird.

7.1 Generierung von Agent Definition Files

In den folgenden Abschnitten wird zuerst analysiert, welche Informationen übertragbar sind, bevor weiter auf die Arbeitsweise des Prototypen eingegangen wird.

7.1.1 Agent Definition Files

In Abschnitt 2.6.1 wurden bereits die ADF vorgestellt (in deren Benutzung führen Pokahr und Braubach, 2004 ein; dort werden auch Beispiele zur korrekten Benutzung der im Folgenden beschriebenen Elemente gegeben). ADF beschreiben die Eigenschaften eines Agenten für die Laufzeitumgebung des Jadex Systems. Beim Start eines Agenten benutzt die Agentenplattform diese XML–Datei(en), um den entsprechenden Agenten zu instanziieren. In Anhang A sind grafische Darstellungen der Struktur dieser Dateien zu finden, die aus der XML Schema[1] Definition generiert wurden[2]. Die Abbildungen A.1 und A.2 zeigen den schematischen Aufbau der Beschreibung eines Agenten. Das Element <*agent*> ist das Wurzelelement.

Agenten können Capabilities enthalten (siehe Abschnitt 2.3). Beschreibungen dieser Capabilities (der selben XML–Schema Definition entsprechend) werden dann im

[1] XML Schema ist eine Standard des World Wide Web Consortium (W3C) Informationen unter: http://www.w3.org/XML/Schema

[2] Namespace–Bezeichner und online verfügbar: http://jadex.sourceforge.net/jadex.xsd

Element <*capability*> referenziert. Die Abbildungen A.3 und A.4 zeigen den entsprechenden Aufbau eines Capability–ADF.

Die Strukturen dieser beiden ADF weisen nur minimale Unterscheide auf. Für Capabilities kann kein Element <*agentdescription*> angegeben werden, die Angabe eines Elementes <*membrane*> für Agenten wird in der derzeitigen Implementation ignoriert. Es wurde in das Schema für zukünftige Entwicklungen aufgenommen. Bis auf diese beiden Ausnahmen, gelten die folgenden Beschreibungen der einzelnen Elemente für beide ADF.

Die Elemente <*imports*>, <*languages*> und <*ontologies*> enthalten lediglich Zeichenketten. In ihnen werden Jadex spezifische Informationen abgelegt. Die <*imports*> enthalten die (voll–qualifizierten Namen) von Java™–Klassen, die in dem (durch den ADF definierten) Agenten benutzt werden. Diese Informationen sind nötig, um es der Laufzeitumgebung zu ermöglichen, die entsprechenden Klassen zu instanziieren. In den <*languages*> und <*ontologies*> können Sprach–Codecs, zur Kodierung von Nachrichten in verschiedenen Formaten und Definitionen von Ontologien abgelegt werden. Diese Codecs und Ontologien werden vom den Klassenbibliotheken des JADE–Systems bereitgestellt. Es wird als Zeichenkette die Anweisung zur Instanziierung des entsprechenden Java™–Objektes angegeben.

Das Element <*properties*> kann eine beliebige Anzahl von <*property*> Elementen enthalten. In einem <*property*> Element werden unter einem Namen (Attribut) Laufzeiteigenschaften eines Jadex–Agenten abgelegt. Diese bestimmen, wie die beschriebenen Agenten im JADE–System ausgeführt werden . Diese Eigenschaften können auch in einer externen XML–Datei abgelegt und mit dem Attribut *properties* des Elementes <*agent*> referenziert werden. Die Angabe dieser Eigenschaften ist optional, es hat sich gezeigt, dass für die meisten Anwendungen die voreingestellte Werte ausreichen.

Es gibt auch die Möglichkeit, Ausdrücke in der Jadex eigenen Anfragesprache zum Zugriff auf die Beliefs eines Agenten, unter einem Namen abzulegen. Diese Ausdrücke können dann von den verschiedenen *conditions* der im Weitern beschriebenen Elemente referenziert werden. Ihre Benutzung ist effizient, da diese Ausdrücke nur einmal eingelesen und compiliert werden müssen. Das Element <*expressions*> kann eine Reihe solcher Ausdrücke in <*expression*>–Elementen beschreiben. Die Angabe eines Namens (Attribut: *name*) ist obligatorisch, optional kann auch ein Typ (Attribut: *type*) angeben werden.

Sowohl Agenten als auch Capabilites können Capabilities enthalten, die wiederum in ADFs beschrieben werden. Das Element <*capabilities*> kann <*capability*>–Elemente enthalten, die die entsprechenden XML–Dateien (Attribut: *file*) unter einem Namen (Attribut: *name*) referenzieren. Diese Referenzen auf Capabilities können eine beliebige Anzahl von <*beliefimport*>–Elementen enthalten. Diese beschreiben mit den Attributen *name* und *from* welche externen Beliefs der Capability bekannt sind.

Das Element <*plans*> enthält Unterelemente <*plan*>, die die einzelnen Pläne referenzieren, die dem jeweiligen Agenten zur Verfügung stehen. Im Attribut *name* wird der Name des Planes angegeben. Das Attribut *instant*, welches die Werte

true oder *false* enthalten kann, gibt an, ob ein Plan bei der Instanziierung des Agenten ebenfalls instanziiert werden soll oder nicht. Das Attribut *priority* kann einen Zahlenwert enthalten, der die Priorität eines Planes für das Laufzeitsystem bestimmt. Der voreingestellte Werlt liegt bei *0*, eine negative Zahl verringert die Priorität, eine Positive erhöht sie entsprechend. In dem Unterelement <*constructor*> wird als Text der Java™–Quellcode zur Instanziierung des entsprechenden Planobjektes (z. B. *new mypackage.MyPlan()*) angegeben. In den Unterelementen <*filter*> und <*waitqueuefilter*> enthalten den Java™–Quellcode zur Instanziierung von sog. Filtern, die die (ACL–)Nachrichten beschreiben. Erfüllt eine Nachricht einen im <*waitqueuefilter*> angegebenen Filter, wird diese an den entsprechenden Plan weitergeleitet. Die sog. passiven Pläne (bei denen das Attribut *instant false* ist) werden durch die Ankunft einer Nachricht, entsprechend des <*filter*>–Elementes istanziiert. Nachrichten die dem Filter im <*waitqueuefilter*> entsprechen, werden für den Plan in der Reihenfolge ihres Eintreffens gespeichert, der Plan kann sie zu einem späteren Zeitpunkt verarbeiten. Diese Elemente sind optional. Außerdem kann optional in dem Unterelement <*condition*> eine Bedingung angegeben werden, die die Ausführung des Planes automatisch startet. Eine Invariante kann in dem Unterlement <*contextcondition*>, eine Vorbedingung in dem Unterelement <*precondition*> angegeben werden. Die Bedingungen sind in einer Anfragesprache, der Object Query Language (OQL) ähnlich, angegeben (ebenfalls von Pokahr und Braubach (2003) beschrieben). Diese beschreibt Zustände der Beliefbase eines Agenten.

Das Element <*beliefs*> kann die Unterelemente <*belief*> und <*beliefset*> enthalten. In den so beschriebenen Beliefs können beliebige Java™–Objekte gespeichert werden. Während ein Belief nur ein Objekt enthält, enthalten Beliefsets eine beliebige Menge von Objekten. Das Attribut *class* gibt an, welche Objekttypen gespeichert werden können. Die Angabe eines Namens (Attribut: *name*) ist obligatorisch. Das Unterelement <*fact*> kann benutzt werden, um (als Zeichenkette) den Code zur Initialisierung eines Objektes (das dann im Belief gespeichert wird) anzugeben. Mit den Attribut *ref* können exportierte (siehe weiter unten) Beliefs aus untergeordneten Capabilities referenziert werden (über ihren Namen). Mit dem Attribut *type* wird die Sichtbarkeit des Beliefs gesteuert, *exportierte* (*type="exported"*) Beliefs können von aussen zugegriffen werden. Im optionalen Attribut *updaterate* kann angegeben werden, in welchen Zeitintervalen die angegebenen Ausdrücke neu ausgewertet werden sollen.

In dem Element <*goals*> werden die einzelnen Ziele eines Agenten aufgelistet. Jadex unterscheidet explizit zwischen drei Arten von Zielen. Ziele die einen bestimmten Zustand herbeiführen sollen(<*achievegoal*>), Zielen die einen Zustand beibehalten sollen (<*maintaingoal*>) und Zielen die lediglich die Ausführung eines Planes anstoßen (<*performgoal*>). Für alle diese Ziele können innerhalb des Elementes <*parameter* > durch <*value*>–Elemente Java™Objekte übergeben werden. Die Erzeugung bzw. das Löschen eines Zieles kann über die Elemente <*creationcondition* >, <*deletioncondition* > und <*targetcondition* > gesteuert werden. In diesen Elementen werden Bedingungen wiederum in der Jadex eigenen Anfragesprache angegeben. Für das <*maintaingoal*> muss zusätzlich eine (<*maintainconditon*>) angegeben werden. In dem Element <*initialgoal*> können (über das Attribut: *ref*) deklarierte Ziele referenziert werden. Sie werden dann direkt bei der Erzeugung eines Agenten/einer Capability instanziiert.

<*servicedescriptions*> enthalten eine beliebige Menge von <*servicedescription*>, die beschreiben, welche Funktionalitäten ein Agent anderen Agenten anbietet. Diese Services werden durch die Attribute *name*, *type* und *ownership* definiert. Zu jedem Service können die Unterelemente <*language*>, <*ontology*> (benutzt wie oben definiert) und <*protcol*> angegeben werden. In dem letztgenannten Element kann der Name eines Protokolles das zur Nutzung des Service verwendet werden soll angegeben werden. Die Angabe von FIPA-spezifizierten Protokollen wird vorgeschlagen, aber die Implementation eines Protokolls ist anwendungsspezifisch.

Diese angebotenen Services werden in dem Element <*agentdescriptions*> in einzelnen <*agentdescription*> gruppiert (wie der Name vermuten lässt nur für die Beschreibung von Agenten). Diese enthalten einen Namen (Attribut) und eine Menge von <*service*>–Elementen, die als Text einen Namen einer definierten <*servicedescription*> enthalten.

Das Element <*membrane*> wird zur Zeit nur für die Definition von Capabilities unterstützt. Es beschreibt Events und Ziele, die eine Capability verarbeiten kann (die an sie weitergeleitet werden), oder die sie an ihre Umgebung (die sie enthaltenden Capabilities oder Agenten) weiterleiten. Empfang oder Weiterleitung werden durch die entsprechenden Elemente <*handles*> und <*posts*> bestimmt. Als ihre Unterelemente sind <*event*> und <*goal*> möglich. Die <*event*>–Elemente enthalten den Java™–Quellcode zur Instanziierung eines entsprechenden Filters, um das Ereignis zu beschreiben. Die <*goal*>–Elemente enthalten den Namen eines unter <*goals*> (siehe weiter oben) definierten Zieles.

7.1.2 Die Objektstruktur des Prometheus Design Tools

Das MetaModell der Objektstruktur eines PDT–Projektes ist umfangreich und wird daher im Anhang B in Abbildung B.1 gezeigt. Ein gesamtes Projekt wird in der Klasse *PrometheusProject* gespeichert. Diese hat Aggregationen zu zwei weiteren Klassen. *ProjectData* enthält zusätzliche Informationen über das Projekt, *PrometheusModelData* enthält alle agentenorientierten Elemente, die modelliert werden. Abbildung 7.1 zeigt das Modell eines einzelnen Agenten. Um eine Vorstellung davon zu vermitteln, wie Agentensysteme im PDT beschrieben werden und welche Informationen ihnen im Hinblick auf Agenten des Jadex Systems entnommen werden können, wird hier die Klassenstruktur eines Agenten in Abbildung 7.1 diskutiert. Der Übersicht halber werden nicht alle Felder jeder Klasse beschrieben.

Zentrales Element ist die Beschreibung eines Agenten ist die Klasse *Agent* (In der Mitte dargestellt; blau hervorgehoben). Ein Agent hat eine Aggregation auf die in ihm enthaltenen Capabilities. Die Modellierung einer Capability entspricht den Agenten. Ihr fehlen lediglich die Assoziationen zu Protokollen (*Protocols*) und Functionalities (*Functionalities*). Protokolle enthalten die ausgetauschten Nachrichten (*includedMessages*) und Beschreibungen, zwischen welchen Agenten diese ausgetauscht werden (*protocolDistributionPairs*). Die Functionalities dienen hauptsächlich der Identifikation der einzelnen Klassen von Agenten während des Entwurfes eines Systems. Sie beschreiben, welche Aufgaben ein Agent unter Zuhilfenahme welcher Aktionen (*Actions*) und mit Zugriffen auf welche Daten (*dataRead*/*dataWritten*) erfüllen kann.

Ein Agent hat Assoziationen zu Zielen (*Goal*), Plänen (*Plan*) und den Daten (*DataConnection*), auf die er zugreifen kann. Die Ziele werden lediglich durch ihre Hierarchie beschrieben. Einziges Element der Klasse *Goal* ist das Feld *subGoals* in dem die jeweiligen Teilziele referenziert werden. Die Pläne enthalten Assoziationen zu den Zielen, die der Plan zu erfüllen sucht, zu den Aktionen, die er ausführen, den Daten auf die er zugreifen, den Ereignissen und Nachrichten, die er verarbeiten und Nachrichten die er versenden kann. Ausserdem werden noch prosaische Beschreibungen der Verarbeitung eines Planes gespeichert.
Die zugreifbaren Daten (*Data*) werden als Klassen modelliert, deren Felder zugreifbar sind. Der Typ der Klasse, ihre zugreifbaren Felder und welche Elemente auf sie zugreifen können, wird dort gespeichert. Agenten sind diese Daten nur über die Klasse *DataConnection bekannt*. Die Semantik der beschriebenen Nachrichten (*Message*) ist leider nicht konform mit der FIPA Spezifikation. In dieser Klasse werden hauptsächlich prosaische Beschreibungen des Inhaltes und Zweckes einer Nachricht abgelegt. Die Klassen *Action* und *Percept* beschreiben Aktionen, die ein Agent ausführen kann und Ereignisse auf die ein Agent reagieren kann. Dies geschieht wiederum durch prosaische Beschreibungen, für die keine Konventionen bestehen.

7.1.3 Übertragbare Informationen

Eine Generierung der ADF aus den Design–Modellen macht nur Sinn, wenn bei den Eingaben in das PDT keine Jadex spezifischen Konventionen gefordert sind, bzw. das Werkzeug im ursprünglichen Sinne weiterverwendet werden kann. Daher wurde sehr genau untersucht, welche der beschriebenen Eigenschaften miteinander kompatibel sind.
Bei den **imports**, **languages**, **ontologies** und **properties** handelt es sich um Eingaben, die spezifisch für Jadex bzw. Jade (languages und ontologies) sind. Ihre Inhalte können aus den Design-Artefakten der Prometheus–Methode nicht gewonnen werden. In Prometheus werden keine Services modelliert, die von Agenten angeboten werden, folglich können auch die Informationen der **agent**- und **servicedescriptions** nicht gefunden werden.
In Prometheus wird sehr anschaulich die Struktur des Agentensystems modelliert. Es wird beschrieben, welche Capabilities (und aus welchen Capabilities diese sich wiederum zusammensetzen), Ziele, Pläne und Beliefs in den Agenten enthalten sind. Die **Capabilities** im ADF werden durch den Dateinamen referenziert, in dem die entsprechende Capability beschrieben wird. Die Dateien werden nach der Namenskonvention *Capabilityname*.xml generiert. Optional können Beliefbases angegeben werden, auf die die Capability Zugriff hat. Im PDT werden die enthaltenen Beliefbases und alle Beliefbases, auf die lesend oder schreibend zugegriffen wird, benannt. Daher können diese (externen) Beliefbases für das ADF extrahiert werden.
Ziele haben in Jadex weit mehr Eigenschaften als im PDT dargestellt. Im werden PDT nur die Hierarchie der Ziele (Teilziele erfüllen Ziele) und eine prosaische Beschreibung der Ziele gespeichert. Es können also alle weiteren, in Jadex unterstützten Eigenschaften (Art des Zieles, etwaige Parameter, initiale Ziele) nicht übertragen werden.
Das Konzept der Filter, um Ereignisse zu beschreiben, ist Jadex eigen. Daher sind die Beschreibungen der **Pläne** kaum kompatibel. Der Konstruktor könnte noch aus einer Namenskonvention ermittelt werden (new *PlanName*()), aber die Filter (bzw. die waitqueuefilter) sind nur aus Default–Werten zu generieren. Im PDT reagieren

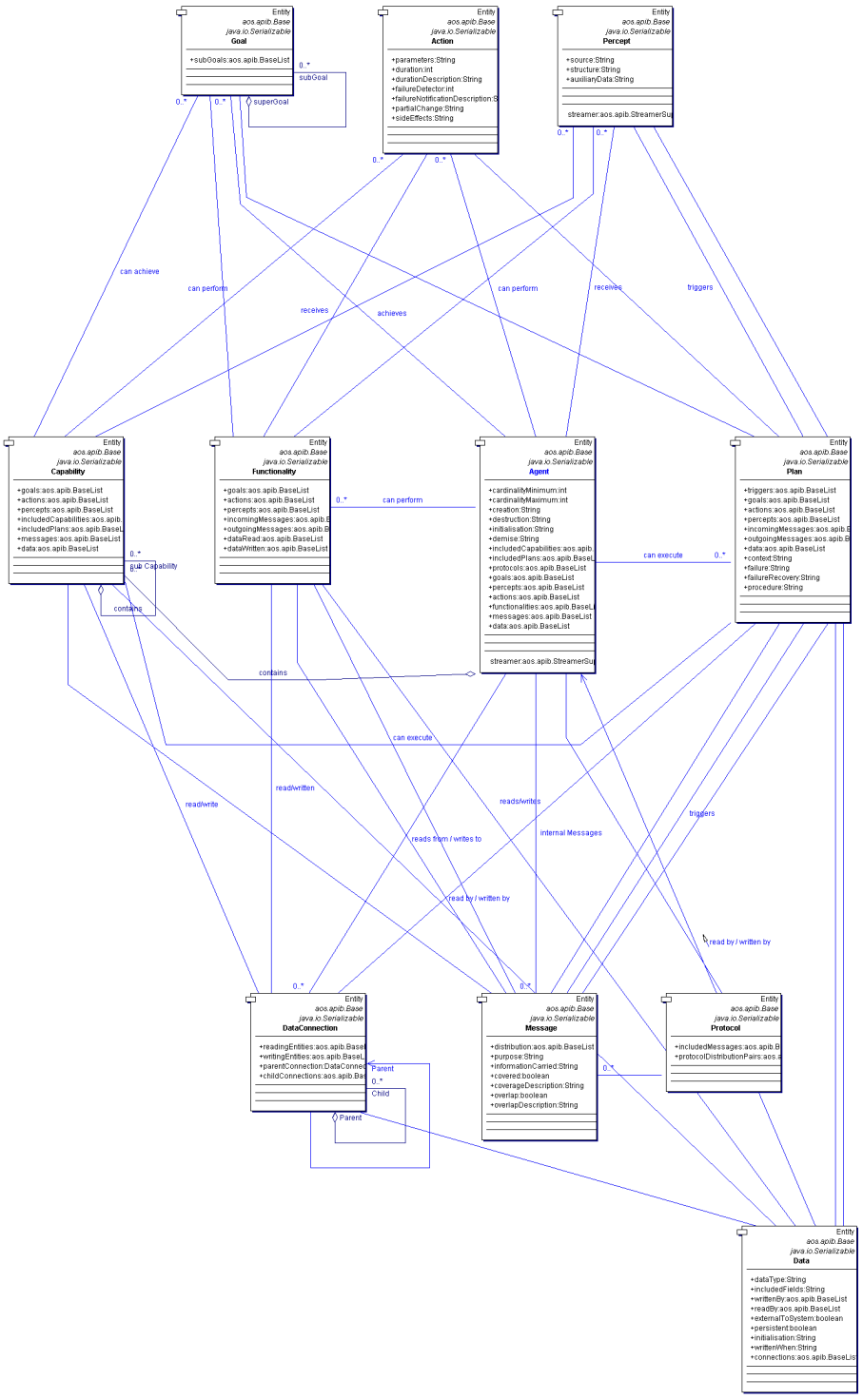


Abbildung 7.1: Model eines Agenten im PDT

Pläne nur auf Nachrichten und Percepts. Auch wenn interne Events (meint Events innerhalb der Agenten) per Konvention durch Percepts beschrieben werden, ist die Abbildung der Goal- und Messageevents kaum möglich. Die Ziele sind in Prometheus nur sehr einfach definiert, ihre Auswirkungen auf die Pläne ist nicht beschreibbar, und die Nachrichten des Systems sind nicht als ACL-Nachrichten repräsentiert. Die Message–Templates von JADE (die Jadex benutzt) filtern aber bewusst aufgrund von ACL–Eigenschaften. Auch die Bedingungen, die Pläne in Jadex triggern können, sind im PDT nicht vorgesehen. Dort werden die Pläne durch die einzelnen Funktionalities eines Agenten getriggert. Dies modelliert, was Pläne erreichen sollen, ist aber für die Implementation in Jadex ohne Belang. Auch die Bestimmung von Plänen, die beim Start eines Agenten instanziiert werden sollen (Attribut: *instant*=true), wird im PDT nicht vorgenommen. Das Attribut *type* kann bei der Erstellung der ADF vernachlässigt werden, es wurde nur zur Kompatibilität mit JADE eingeführt, und wird erfahrungsgemäß kaum benutzt.
Auch die Modellierung der **Beliefs** ist unterschiedlich. In den ADF wird eine Sammlung verschiedener, einzelner Beliefs angegeben, die unterschiedliche Objekte speichern können (Auch der Code zur Initialisierung dieser einzelnen Beliefs wird angegeben). Im PDT wird als Beliefbase eine Klasse angegeben, in der die Beliefs gespeichert sind, sowie die zugreifbaren Felder dieser Klasse und eine prosaische Beschreibung der Initialisierung. Anstelle der Sichtbarkeitsregeln der Beliefs (innerhalb der Capabilities) kann im PDT nur angegeben werden, ob eine externe Datenquelle (extern im Sinne von nicht innerhalb eines Agenten) benutzt wird. Da alle Pläne eine Liste der gelesenen und geschrieben Daten und der enthaltenen Beliefbases haben, kann auf diesem Wege auf die exportierten (Attribut: *type="exported"*; bzw. referenziert durch das Attribut *ref="BeliefName"*) Beliefs geschlossen werden.
Im PDT werden die eingehenden und ausgehenden Nachrichten, sowie die eingehenden Percepts, einer Capability, aufgelistet. Diese können benutzt werden, um sie in der **Membrane** zu beschreiben. Allerdings gelten hier die selben Einschränkungen, wie sie oben für die Beschreibung der Pläne festgestellt wurden. Die entsprechenden Filter sind aus den Informationen, die im PDT gespeichert sind, nicht abzuleiten. Im PDT senden Capabilites lediglich Nachrichten nach außen. Den einfachen Beschreibungen der Ziele eines Agentensystems ist nicht zu entnehmen, ob sie nach außen sichtbar sein sollen, oder in einer Capability bearbeitet werden.

Es wurde gezeigt, dass die Struktur eines modellierten Agentensystems in die ADF von Jadex umformuliert werden kann. Aber viele Informationen, die in ADF enthalten sind, sind spezifisch für das Jadex System und können deshalb kaum aus den Designartefakten gewonnen werden.

7.2 Der entwickelte Prototyp

Zu Beginn der Untersuchung war es das Ziel, eine weitgehend automatische Transformation von PDT–Projekt–Dateien in ADF zu ermöglichen. Die obigen Abschnitte haben aufgezeigt, dass dies nicht zweckmäßig sein kann. Daher wurde ein Prototyp entwickelt, der diese Projektdateien benutzt, um die Erstellung der ADF so weit wie möglich zu unterstützen. Es wird ausgelesen, was möglich ist, der Benutzer kann die zu erstellenden ADF auswählen und spezifiziert in einer grafischen Benutzungsoberfläche, die für Jadex spezifischen Eigenschaften. Die grafisch geführte Editierung dieser XML-Dateien soll nicht nur deren Erstellung erleichtern, sondern insbeson-

dere unerfahrene Benutzer unterstützen. Zusätzlich soll der Aufbau eines Agentensystems mit den Mitteln der Prometheus–Methode ausgedrückt werden. Daher wird ein Mechanismus entwickelt, um eine Menge von ADF in eine PDT–Datei zu übersetzen. Dabei ist ein Informationsverlust (wie im vorigen Kapitel dargestellt) unvermeidlich. Derzeitige Quellcode–Editoren können, durch die Interpretation einer XML–Schema Definition, die Syntax eines entsprechenden XML–Dokumentes überprüfen und die Eingaben des Benutzers auf gültige Werte beschränken. Durch die interne Repräsentation der ADF gehen die Möglichkeiten des entwickelten Prototypen darüber hinaus, die Konsistenz von Referenzen innerhalb des Dokumentes kann gewahrt werden.

7.2.1 Verwendete Werkzeuge

Das Programm wurde mit Java™(1.4.2_01)/Swing entwickelt. Für die Entwicklung der Benutzeroberfläche wurde die *Eclipse*–Entwicklungsumgebung[3] (1.2.1) mit dem *Eclipse Visual Editor* Plug–in[4] (0.5.0RC2) benutzt. Dies ist eine leicht zu bedienende Umgebung zur Erstellung von Benutzungsoberflächen.

JACOB Object Modeller

Das PDT wurde am RMIT in Zusammenarbeit mit der Firma Agent Oriented Software Pty. Ltd. entwickelt. Es verwendet deren *JACOB Object Modeller* zur Speicherung (in einem XML–Format) und Initialisierung von Projekt–Dateien, die ein Agentensystem beschreiben. Dankenswerterweise gab diese Firma die Erlaubnis zur Benutzung dieses Werkzeugs für den Prototyp. Die genaue Funktionsweise ist für diese Arbeit unerheblich. Im Folgenden wird ein Eindruck von der Funktionalität und der Benutzung gegeben. Die Dokumentation von Agent Oriented Software Pty. Ltd. (2003) beschreibt detailliert die Verwendung dieses Werkzeuges.

Es handelt sich bei JACOB um ein Werkzeug für die Initialisierung und den Transport von Objekten zwischen verschiedenen Programmiersprachen (ähnlich dieser Funktionalität von CORBA). Hierzu werden Objektstrukturen in sog. *dictionary*–Dateien (durch die *JACOB Data Definition Language*) beschrieben. Es existiert ein grafisches Werkzeug (*JACOB Objekt Browser*), das die Editierung dieser Beschreibungen unterstützt. Die Ausgabe und Initialisierung der beschriebenen Strukturen ist aus vier Formaten möglich: ASCII, binär, per JDBC und XML. Das PDT benutzt nur die letzte Speicherungsform.

Die vom PDT verwendeten Dictionary-Dateien wurden für diese Arbeit vom RMIT zur Verfügung gestellt. Nachdem ein Programm diese Beschreibungen eingelesen hat, können durch einen einfachen Methodenaufruf die gespeicherte Klassenstruktur eingelesen werden. Bei dem eingelesen Objekt handelt es sich dann um eine Instanz der Klasse `au.edu.rmit.cs.prometheus.datamodel.jacob.Prometheus Project` (siehe Anhang B).

[3] Siehe: http://www.eclipse.org/org/index.html
[4] Detaillierte Beschreibung unter: http://www.eclipse.org/vep/

Velocity Template Engine

Um eine Anpassung an zukünftige Veränderungen am Aufbau der ADF zu vereinfachen, beruht die Generierung der ADF–Dateien auf austauschbaren *Templates*. Eine Reihe von sog. *Template Engines* existieren, die mittels der Java™–Programmier–sprache benutzt werden können. In deren Benutzung führen eine Reihe von Internetseiten ein. Beispielhaft sei hier auf einen Artikel von Messerschmidt (2001) verwiesen, der in die Benutzung einführt. Nach einer kurzen Recherche wurde die *Velocity Template Engine* (1.3.1) eingesetzt. Hierbei handelt es sich um ein Open Source Projekt der Apache–Jakarta–Gruppe[5]. An der Universität Hamburg wurde diese des öfteren eingesetzt und hat sich bewährt (in die Benutzung führt Apache Jakarta Project 2003a ein). Die Arbeitsweise dieses Systems wird in Abbildung 7.2 dargestellt. Beliebige Objekte können in einem Context–Objekt der Template Engine

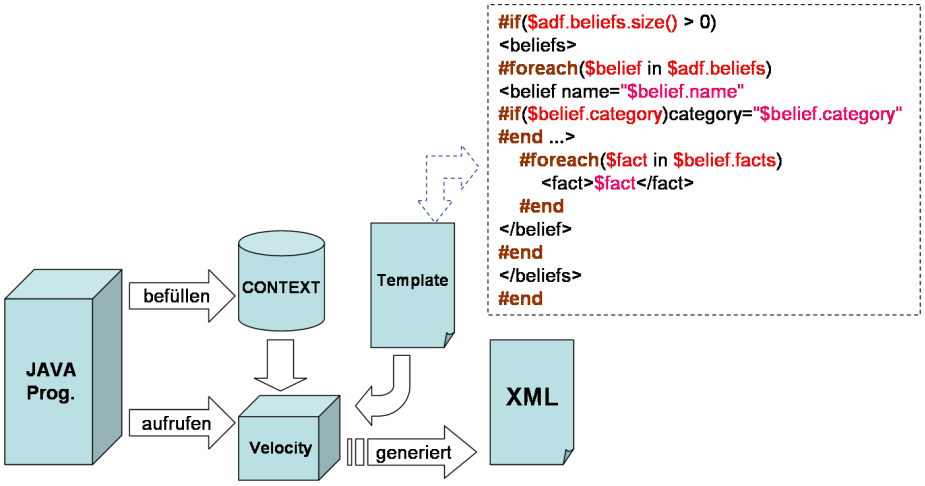


Abbildung 7.2: Arbeitsweise der Velocity Template Engine

(`org.apache.velocity.VelocityContext`) gespeichert werden. Eine Einführung in Benutzung und Darstellung der Anforderungen an auslesbare Objekte sind in der Dokumentation des *Apache Jakarta Project* (2003b) zu finden. Beim Aufrufen der Engine wird eine Template–Datei angegeben. Diese enthält beliebigen Text mit Referenzen auf Felder von Objekten, die im Context–Objekt erreichbar sind. Diese Referenzen werden in der *Velocity Template Language* (VTL) angegeben, die einfache Kontrollstrukturen zur Steuerung der Ausgabe bereitstellt.
Die Abbildung 7.2 zeigt einen Ausschnitt aus einer solchen Template. Farblich hervorgehoben sind die Kontrollstrukturen. In dem Beispiel wird das Feld *beliefs* der Klasse *adf*, das eine Collection enthält, nach seiner Größe abgefragt. Dann wird diese iteriert und in jedem Iterationsschritt ein Element <belief >in die die Ausgabe geschrieben. Hierbei wird das Feld *category*, der in der Collection gepeicherten Elemente, als Attribut eingefügt. Nach diesem Verfahren werden die ADFs generiert. Im Context wird hierzu ein Objekt abgelegt, dass das Interface *jadex.tools.pdt2adf.wrapper.interfaces.IAdf* implementiert. Dieses beschreibt die Elemente eines ADF und wird über die Benutzeroberfläche verändert. Dieses Objekt erlaubt die objektori-

[5]Informationen unter: http://jakarta.apache.org/velocity/index.html

entierten Zugriffe auf Felder, wie sie in dem Beispiel–Template der Abbildung 7.2 erkennbar ist.

7.2.2 Beschreibung des Prototypen

Die grundlegende Arbeitsweise ist dreigeteilt. Eine Reihe von Klassen ist für Darstellung und Steuerung der Benutzeroberfläche (`jadex.tools.pdt2adf.gui`) zuständig. Diese manipulieren die Felder einer objektorientierten Repräsentation der Struktur eines ADF (`jadex.tools.pdt2adf.model`) und reagieren auf Änderungen in den Eigenschaften dieser Repräsentation. Zudem existiert eine kleine Anzahl von Hilfsklassen(`jadex.tools.pdt2adf.helpers`), die für weitere Aufgaben, wie Einlesen und Speichern der PDT-Projekte/XML–Dateien und Konvertierungen, verantwortlich sind. In diesem Package ist auch das von der oben beschriebenen Template Engine verwendete Template abgelegt. Es wurde in Abschnitt 7.2.1 dargestellt, wie diese Templates auf einem Objekt beruhen, deren Felder in der verwendeten Template–Sprache abgefragt werden können. Die Repräsentation der ADF wird in einem eigenen Interface (`jadex.tools.pdt2adf.wrapper.interfaces.IAdf`) beschrieben. In diesem Package befinden sich auch weitere Interfaces, die die Zugriffe auf die einzelnen Elemente eines ADF definieren.
Im Jadex System gibt es bereits eine eigene, interne Repräsentation der Struktur der ADF–Dateien (`jadex.model`). Diese wird benutzt, um beim Start eines Agenten, dem Laufzeitsystem die angegebenen Werte zur Verfügung zu stellen. Beim Instanzieren dieser Strukturen werden die Werte eingelesenen und überprüft. Es existiert ein weiteres Package (`jadex.tools.pdt2adf.wrapper`), das einen (Klassen–)Adapter (wie beschrieben von Gamma et. al, 1997), für diese Modellstruktur bereitstellt. Diese Package–Struktur ist grafisch in Anhang C dargestellt.
In der Benutzungsoberfläche kann die Art des intern verwendeten Modells (validierend per Adapter oder nicht) umgeschaltet werden.

Abbildung 7.3 zeigt das Hauptfenster nach dem Start des Prototypen. Es zeigt die Anwendung, nachdem das bereits aus Abschnitt 6.4 bekannte Beispiel–Projekt, eingelesen wurde. Auf der linken Seite werden die in dem Projekt enthaltenen Agenten und Capabilites (welche durch ADF beschrieben werden) in einer Baum–Struktur angezeigt. Werden diese selektiert, werden ihre Eigenschaften in den Formularfeldern in der Mitte des Anwendungsfensters angezeigt. Diese sind, wie auch einige andere (für das Jadex System spezifische) Eigenschaften, über das sichtbare Popup–Menu erreichbar. Dieses Menü variiert, je nachdem ob ein Agent oder eine Capability beschrieben wird. Auf der rechten Seite werden Möglichkeiten zum Einlesen einer Projekt–Datei, zum Speichern von ADF-Dateien, zur Auswahl einer Template–Datei und zum Speichern einer PDT–Projekt–Datei gegeben. Mit dem Auswahlfeld unter den Schaltflächen kann kontrolliert werden, ob das validierende Modell des Jadex Systems für die Repräsentation der ADF oder ein nicht überprüfendes Modell benutzt werden soll. Wird das überprüfende Modell verwendet, werden nur gültige Namen und Ausdrücke in der Jadex eigenen Anfragesprache akzeptiert. Außerdem müssen für alle referenzierten Pläne kompilierte Java™–Dateien vorliegen, die über den Klassenpfad (Angabe von benötigten import–Anweisungen) erreichbar sein müssen.

Es wurde in Abschnitt 7.1.1 die Vielzahl von Eigenschaften dargestellt, die an einzelnen Elementen (wie Zielen, Plänen, etc.) spezifiziert werden können. Die einzelnen

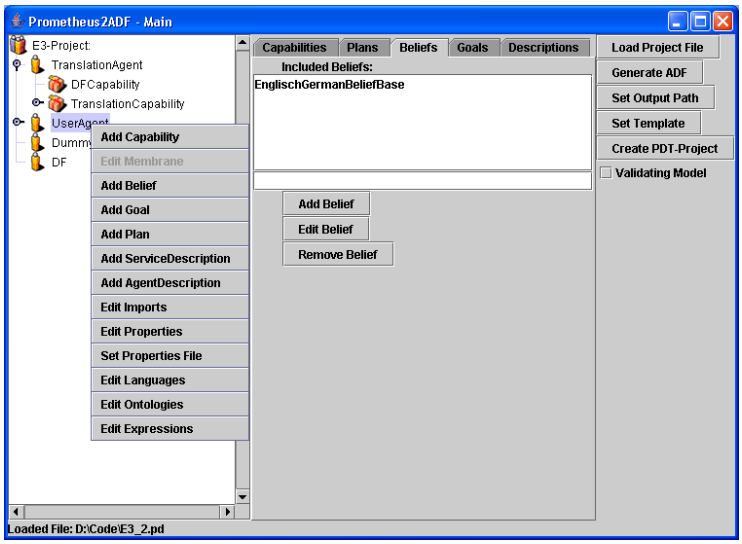


Abbildung 7.3: Bedienungsoberfläche des Prototypen

Elemente eines ADF werden über Dialoge gesteuert. Abbildung 7.4 zeigt einen Dialog zur Editierung eines Zieles. Die verschiedenen Eigenschaften des Meta-Level-Reasoning können über Checkboxen eingestellt werden. Ein Selektor erlaubt die Auswahl der Art von Ziel (wie beschrieben in Abschnitt 7.1.1), die wiederum beeinflusst, welche Eigenschaften auswählbar sind. In entsprechender Weise werden die übrigen Elemente editiert und die Konsistenz der Eingaben gewahrt. Beispielsweise sind in dem Dialog zur Editierung der Agentdescriptions (der eine Menge von Services enthält, siehe Abschnitt 7.1.1) eines Agenten nur bereits definierte Services auswählbar. Bemerkenswert ist noch der Dialog zur Erstellung eines PDT-Projektes aus einer Sammlung von ADF–Dateien. In Abbildung 7.5 wurden zwei XML–Dateien ausgewählt und eine Projekt–Datei aus ihnen generiert. Es werden hierbei alle beteiligten ADF, auch diejenigen der enthaltenen Capabilities angegeben. Die Namen der beschriebenen Agenten und Capabilites müssen eindeutig sein. Die Angabe von Kommentaren, wie im gezeigten Dialog vorgesehen, ist möglich und wird im Projekt gespeichert. Der Algorithmus zur Erstellung des Projektes besteht zur Zeit aus drei Phasen. Die Erstellung eins Projektes besteht hauptsächlich darin, (1) die spezifizierten Elemente einzulesen, deren Eigenschaften soweit möglich (siehe Abschnitt 7.1.3) zu konvertieren und sie dem Projekt hinzuzufügen. Dann (2) werden die Elemente mit den entsprechenden Agenten und Capabilities referenziert. Außerdem werden die Elemente (3) dem System Overview Diagram hinzugefügt. Offensichtlicher Weise sind keine Informationen über die Anordnung der Agenten im System aus den ADF erkennbar, daher sind die so generierten Diagramme nur wenig aussagekräftig. Der Benutzer muß die Diagramme anordnen und die Protokolle zwischen den Agenten spezifizieren.

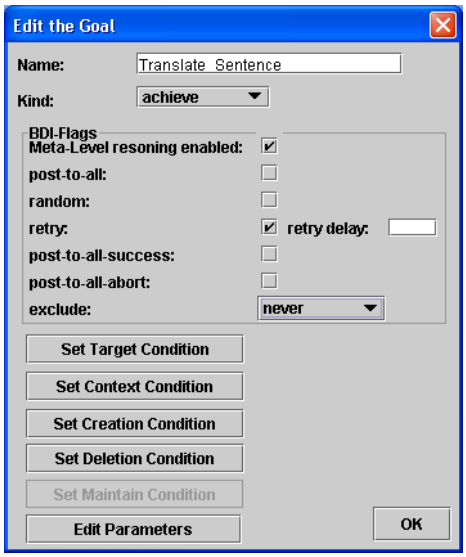


Abbildung 7.4: Dialog zur Definition eines Zieles

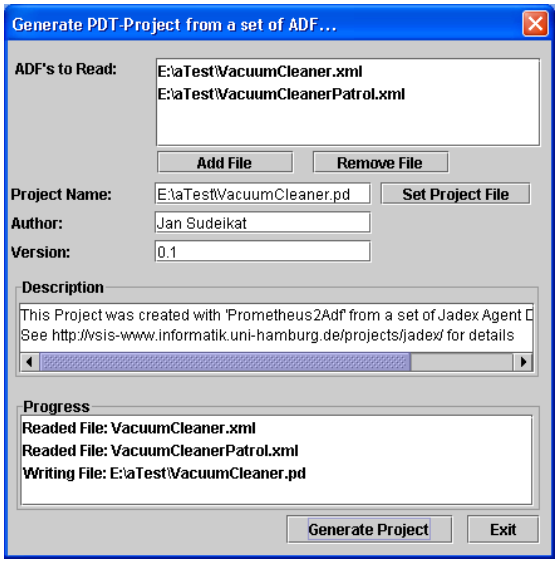


Abbildung 7.5: Dialog zur Erstellung eines Projektes aus einer Sammlung von XML-Dateien

7.3 Anwendung der Prometheus–Methode und des Werkzeugs

Im Verlauf dieser Arbeit wurde die Prometheus–Methode als am geeignetsten zur Unterstützung der Anwendungsentwicklung mittels Jadex gefunden und ein Werkzeug vorgestellt, mit dem Modelle dieser Methode in ADF des Jadex Systems übersetzt werden können. In diesem Abschnitt wird nun eine kleine Anwendung mit dem Jadex System, unter Verwendung der gefundenen Methode, dem PDT und dem entwickelten Prototypen, erstellt. Diese beispielhafte Entwicklung gibt einen Eindruck davon, wie die Entwicklung von Anwendungen unterstützt wird.

7.3.1 Das zu implementierende System

Ferber (2001) beschreibt eine Reihe von Anwendungsgebieten für Agentensysteme. Hierbei führt er verschiedene Arten von Simulationen an. Insbesondere die Simulation und/oder Steuerung von Robotern nennt er als offensichtliches Aufgabengebiet, das oft von Forschern aufgegriffen wird. Die Metapher des Roboters ist anschaulich und eingängig, zur Beschreibung von Agenten.
Um verschiedene Konzepte zu beschreiben, verwendet er als Beispiel eine Gruppe von Robotern, die autonom auf dem Mars Aufgaben erfüllen sollen (ebd.:64). Dies ist Science–Fiction, aber Ferber beschreibt wie diese Aufgabenstellung von Forschern in Zusammenarbeit mit der NASA angegangen wurde (ebd.: 65)[6].
Es ist offensichtlich, dass diese Roboter autonom ihre Aufgaben erfüllen müssen, da die Entfernung zur Erde zu groß ist, um eine direkte Steuerung von dort zu erlauben.
Im Zusammenhang dieser Arbeit wurde ein abgewandeltes Szenario programmiert, das eine feste, vordefinierte, hierarchische Organisationsstruktur (*feste Teamorganisation*) von Agenten beschreibt (ebd.:143). Die statischen Rollen und Beziehungen unter ihnen sind im Voraus definiert. Diese Organisation besteht aus drei Arten von Robotern (gezeigt in Abbildung 7.6). Einem Roboter der mit speziellen Sensoren

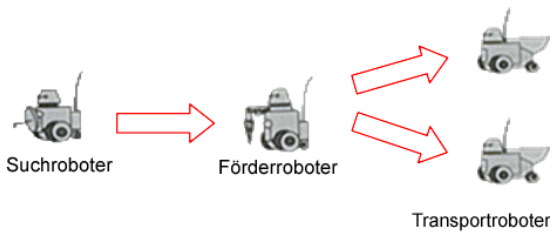


Abbildung 7.6: Die Organisation der Agenten (angelehnt an Ferber 2001:143)

ausgestattet ist, um Erze im Mars–Gestein aufzuspüren, einem Roboter der speziell für die Förderung von gefundenen Erzen ausgestattet ist, und einer variablen Anzahl von einfachen Robotern, die für den Transport des geförderten Erzes zur Heimat Basis verwendet werden. Die Suchroboter kontrollieren die Förderroboter, da diese von ihnen abhängig sind. Ohne gefundene Erzvorkommen können sie Ihre Aufgabe nicht erfüllen. Die Förderroboter wiederum kontrollieren die Transportroboter.

[6] Außerdem wecken aktuelle amerikanische Weltraum–Pläne Interesse für dieses Szenario, siehe (Bush, 2004)

Es ist zu bemerken, dass das beschriebene Szenario eine klassische Anwendung für reaktive Agenten darstellt. Die Aufgaben der Agenten können in diesem überschaubaren Beispiel sequentiell abgearbeitet werden, was den Einsatz von dieser Art Agenten, die lediglich auf Ereignisse reagieren, nahe legt. Insbesondere zur Entwicklung von Robotern wurden diese Architekturen oft vorgeschlagen (wie z. B. Nielsen, 1998 beschreibt). In der Implementation wird das Verhalten der Agenten durch jeweils einen Plan festlegt, der die Hierarchie der verschiedenen Ziele definiert. Nach dem Erfüllen eines Zieles erzeugen diese Pläne neue Teilziele, bis die Aufgabe gelöst wurde. Dieser Mechanismus, Aufgaben sequentiell abzuarbeiten und mit der Bearbeitung der nächsten Aufgabe erst zu beginnen, wenn die vorigen erfüllt wurden, kann auch durch reaktive Agentenarchitekturen implementiert werden (als Beispiel sei hier die Subsumption-Architektur (kurz vorgestellt in Abschnitt 2.3; Brooks, 1990). Diese Szenario wurde hauptsächlich wegen seiner Anschaulichkeit und Übersichtlichkeit gewählt. Die sequentielle Arbeitsweise wird hier durch die bedingte Erzeugung von Teilzielen modelliert.

7.3.2 Die Anwendung

Abbildung 7.7 zeigt einen Ausschnitt aus der laufenden Simulation. Die gestrichelten Pfeile wurden nachträglich in das Bild eingefügt, um die Bewegungen der Agenten (Roboter) zu verdeutlichen. Die roten Kreise bezeichnen Positionen, an denen Erz vermutet wird.
In der rechten oberen Ecke des Spielfeldes bewegt sich der Suchroboter auf eine mögliche Produktionsstelle zu. Wenn es an ihr möglich ist Erz zu fördern, wird der Agent dem Förderroboter eine Repräsentation dieser Position zuschicken. In der linken unteren Ecke des Spielfeldes ist ein grau hinterlegter Kreis zu erkennen. Er bezeichnet eine Position, die der Suchroboter bereits untersuchte, an der aber kein Erz gefördert werden kann.
In der rechten unteren Ecke befindet sich ein Förderroboter neben einem Kreis an dem *100%* aufgetragen ist. Diese Prozentzahl zeigt die Menge an Erz, die an der Stelle gefördert wurde. Wird der Förderroboter an eine Position gerufen, beginnt der mit der Förderung des Erzes. Bei einem Stand von *100%* wird der die Förderung stoppen und einen Transportroboter auffordern, das Erz abzutransportieren. Danach ist er bereit, auf weitere Anfragen des Suchroboters zu reagieren.
In der Mitte des Spielfeldes sind zwei Transportroboter mit dem Abtransport von Erz beschäftigt. Beim Transport pendeln die Roboter zwischen der Förderstelle (die ihnen wiederum von Transportroboter mitgeteilt wurde) und der Startposition (linke obere Ecke) hin und her, bei jeder Fahrt kann eine, im Agenten definierte Menge von Erz befördert werden. Der obere dieser beiden Transportroboter bewegt sich auf eine Position mit dem Füllstand von 80% zu, was bedeutet, dass er bereits 20% abtransportiert hat. Wurde an eine Position vollständig abgebaut, bleiben die jeweiligen Förderroboter an der Startposion.

7.3.3 Modellierung des Systems

Hier wird kurz dargestellt, wie das eben beschriebene System in Prometheus modelliert werden kann.

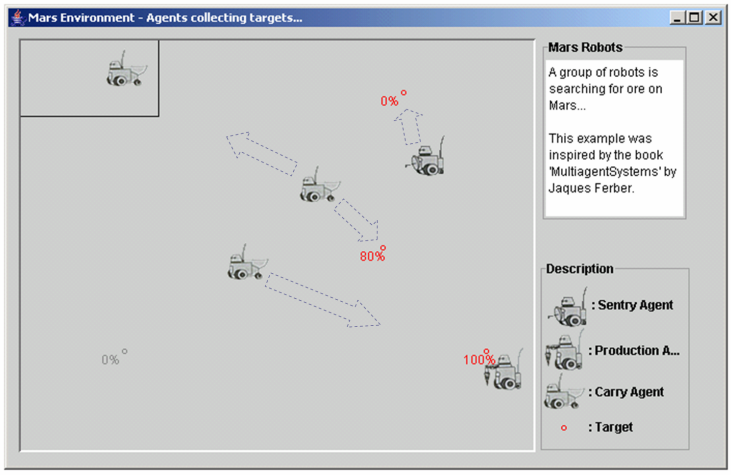


Abbildung 7.7: Die Beispielimplementation

Die Arbeitsweise wurde bereits beschrieben, daher wird auf die Modellierung von Szenarien, um die Anforderungen an das Systems zu finden, nicht weiter eingegangen. Die Ziele des Systems sind in Abbildung 7.8 dargestellt. Das Ziel des Systems

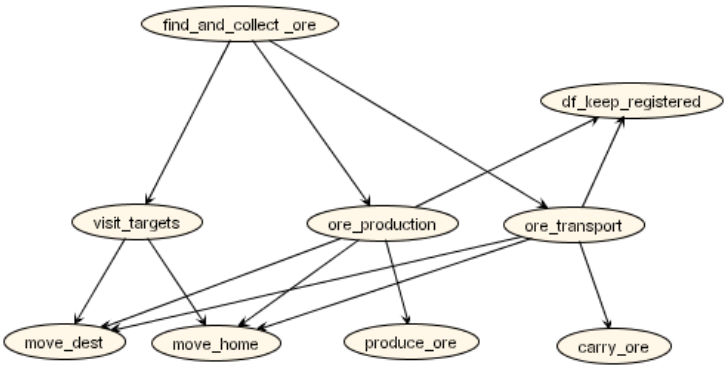


Abbildung 7.8: Die Ziele des Systems

ist das Sammeln von Erzen (*find_and_collect_ore*). Um dies zu tun, muss erst dessen Vorhandensein an den vorgegebenen Positionen überprüft werden (*visit_targets*). Außerdem muss das gefundene Erz an den Positionen gefördert (*ore_production*), und dann abtransportiert werden (*ore_transport*).
Diese Ziele werden, durch die im vorigen Abschnitt beschriebenen Agenten, erfüllt. Ihre Aufgaben können wiederum in Teilzielen gegliedert werden. Damit der mit der Suche beauftragte Agent von den anderen Agenten die Erfüllung von Aufgaben anfragen kann, müssen diese sich am DF des Systems anmelden (*df_keep_registered*). Für alle Agenten ist es notwendig, sich an bestimmte Positionen zu Bewegen (*move_dest*) und auch wieder zu ihrem Ausgangspunkt zurückzukehren (*move_home*). Die eigentliche Förderung des Erzes ist als eigenes Teilziel modelliert. Zum Abtrans-

port muss das Erz geladen und wieder entladen werden. Dies wird in einem eigenen Ziel (*carry_ore*) dargestellt. Die letztgenannten Ziele erfordern eine direkte Interaktion mit der Umgebung. Diese Einwirkungen auf die Umgebung werden in Prometheus in den sog. Actions beschrieben. Diese wiederum dienen der Ausführung von Functionalities, die beschreiben, welche Aufgaben ein Agent erfüllen kann. Abbildung 7.9 zeigt die Zuordnung der Actions zu diesen Zielen. Um den Zielen die Aktionen zu-

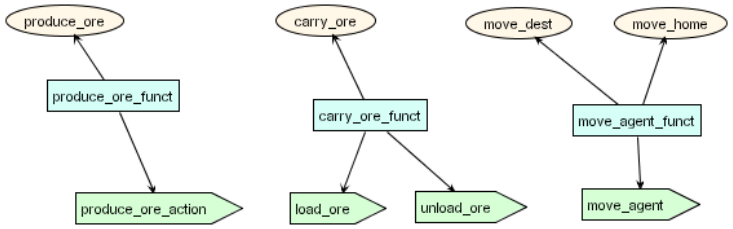


Abbildung 7.9: Die Zuordnung von Actions zu den Zielen

zuordnen, wurden gleichlautende Functionalities modelliert. In der späteren Implementation werden diese Aufgaben durch die Erstellung von Teilzielen erfüllt (siehe oben) . In einer Modellierung, für ein System, das diesen Mechanismus nicht kennt, würde man auf die Ziele in der untersten Ebene verzichten und diese direkt als Functionalities beschreiben. In der Zusammenstellung ist ersichtlich, dass die Tätigkeit Erz zu transportieren (*carry_ore_funct*) durch Aufrufen der Aktionen be- und entladen (*load_ore* / *unload_ore*) ausgeführt wird (Der eigentliche Transport erfolgt über die Bewegung des Agenten durch *move_home*). Die Bewegungen der Agenten erfordern nur eine Aktion, das eigentliche Bewegen des Agenten (*move_agent*).

Aus der Aufgabenstellung sind die verschiedenen Arten von Agenten vorgegeben. Daher ist das Vorgehen, wie es in Prometheus vorgeschlagen wird, um die Agenten zu identifizieren, hier unangebracht. Aus den Functionalities kann hier nicht unmittelbar auf die einzelnen Klassen von Agenten geschlossen werden. Dazu haben die Agenten in diesem Beispiel zu viele ähnliche Eigenschaften.

Die Identifikatin der Agenten führt direkt zum Übersichtsdiagramm des Systems (System Overview Diagram) in Abbildung 7.10. Hier sind die drei modellierten Arten von Agenten und ihre Aktionen eingetragen. In der oberen linke Ecke ist ein weiter Agent eingezeichnet (*Environment*). Er ist lediglich für die Darstellung und Speicherung der Umgebung der Agenten verantwortlich. Daher steht er in keiner agentenbasierten Kommunikationsverbindung mit den anderen Agenten. Um atomare Beeinflussungen der Umgebung (Produktion von Erz an einer Position, Sensordaten über die Erzvorkommen an einer Position) zu simulieren, wird das Modell der Umgebung über statische Methoden verändert. Die Darstellung der Welt erfolgt nur aus implementationstechnischen Gründen über einen Agenten. Er wird daher nicht weiter betrachtet. In diesem Modell wird nun auch die Kommunikation zwischen den Agenten dargestellt. Es sind zwei Protokolle eingezeichnet(*request_to_produce_at_location* und *request_to_carry_ore_to_homebase*). Das Abbildung 7.10 zeigt eine Auflistung der Nachrichten eines Protokolls, wie sie im PDT angezeigt werden.

Die einzelnen Agenten werden in den den entsprechenden Übersichtsdiagrammen

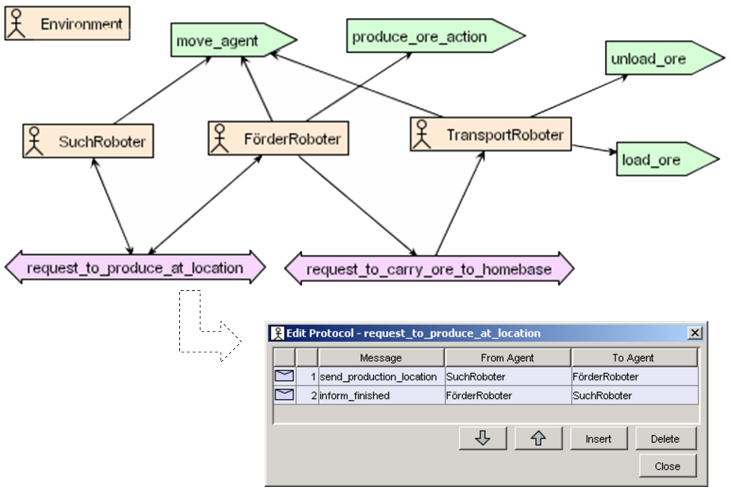


Abbildung 7.10: System Overview Diagram der Anwendung

(Agent Overview Diagram) dargestellt. Exemplarisch wird der Förderroboter in Abbildung 7.11 beschrieben. Er zeigt das komplexeste Verhalten. Bei Erhalt der Nach-

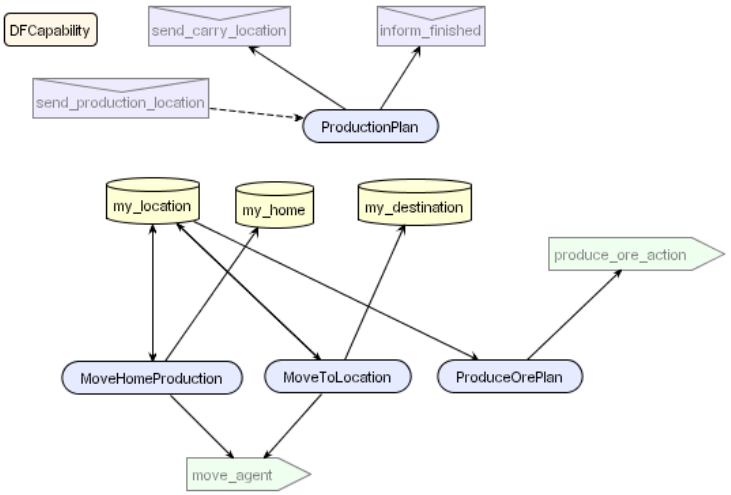


Abbildung 7.11: Übersicht über den Förderroboter

richt *send_production_location* wird der Plan *ProductionPlan* gestartet (im Deskriptor eingetragen im Feld *Triggers*). Diese Nachricht stammt von Suchroboter und enthält ein Objekt (von der Klasse `jadex.tools.pdt2adf.example.Location`), welches eine Position innerhalb der simulierten Umgebung repräsentiert, in der ein Erzvorkommen gefunden wurde. Dieser Plan kann wiederum die Nachrichten *inform_finished* (um dem Suchagenten anzuzeigen, das eine beauftragte Produktion beendet wurde) und *send_carry_location* (Um einen Transport-Agenten damit zu beauftragen das Produzierte Erz von einer bestimmten Position abzutransportie-

ren) absenden. Die weiteren Pläne, die in dem Agenten existieren, werden wie oben beschrieben durch die Erstellung von Teilzielen gestartet. Sie sind für die Bewegung des Roboters (*MoveHomeProduction* / *MoveToLocation*) und die eigentlich Förderung des Erzes (*ProduceOrePlan*) zuständig. Atomare Aktionen können den Plänen jeweils zugeordnet werden (wie die Aktion *produce_ore_action*). Es ist auch ersichtlich, auf welche Beliefbases die einzelnen Pläne zugreifen (*my_location*, *my_home* und *my_destination*; die Richtung der Pfeile gibt an ob es sich um lesenden und/oder schreibenden Zugriff handelt). In der linken oberen Ecke ist eine Capability (*DF-Capability*) eingezeichnet. Sie ist zuständig für das Registrieren und Auffinden von Agenten im DF des Systems. Sie wird von der Klassenbibiliothek des Jadex Systems gestellt und daher hier nicht weiter betrachtet.

7.3.4 Betrachtung der Entwicklung / Unzulänglichkeiten

Die Modellierung dieses Systems verdeutlicht die bereits festgestellten Unzulänglichkeiten der Modellierungsmethode.[7] Die Modellierung innerhalb der Pläne ist undeutlich. In den Plänen wird das sequentielle Verhalten, das die Agenten an den Tag legen, durch die Erstellung neuer Teilziele innerhalb eines Planes realisiert. Die Klassenbibiliothek, die das Jadex System zur Verfügung stellt, erlaubt dies. Allerdings ist dieser Mechanismus in der Prometheus–Methode nicht darstellbar.
In den einzelnen Übersichtsdiagrammen zu den Agenten sind die Pläne dargestellt. Der Ablauf der Pläne, in diesem Beispiel vor allem das Warten auf Nachrichten oder Ereignisse, wird nicht modelliert. Es wird lediglich gezeigt, welche Nachrichten und Ereignisse von Plänen verarbeitet werden. Außer bei Plänen, die direkt mit dem Erhalt einer Nachricht gestartet werden, ist nicht erkennbar, was zu ihrer Ausführung führt. Die Anzahl der einzelnen Agenten wird nur in den zugehörigen Deskriptoren festgehalten. Eine optionale Anzeige mehrerer Agenten gleichen Typs im Diagramm wäre wünschenswert und würde in diesem überschaubaren Beispiel die Zusammenarbeit verdeutlichen. Erst durch ein Nachschlagen in den Deskriptoren ist ersichtlich, dass es mehrere Transportroboter geben kann.
Außerdem wird in der Modellierung darauf deutlich gezeigt, welche Tätigkeiten Agenten ausführen können (durch die Funktionalities), Ziele die Agenten gemein sind, sind aber aus den Diagrammen nicht erkennbar. Außerdem ist nicht darstellbar, wie Agenten zur Erfüllung von Zielen beitragen. Es wird nur dargestellt, ob sie dies tun.

Nach der Modellierung des Systems sind die vier zu erstellenden ADF in der Baumstruktur des Prototypen vorgegeben (Abbildung 7.12). Alle in diesen ADF beschriebenen Beliefs, Ziele, Pläne und Capabilities sind in der Benutzeroberfläche eingetragen und mit voreingestellten Werten gefüllt. Die für Jadex spezifischen Elemente müssen vom Benutzer eingestellt werden. Im nicht validierenden Modus sind jegliche Texte eingebbar, diese werden dann in den entsprechenden XML–Elementen eingesetzt. Wird das validierende Modell von Jadex benutzt, ist die Benutzung unbequemer. So müssen zum Beispiel alle referenzierten Pläne kompiliert und über den Klassenpfad des Systems erreichbar sein.
Um insbesondere unerfahrene Benutzer in der Erstellung der ADF anzuleiten, sind eine Reihe von Erweiterungen der Benutztungsoberfläche denkbar. So könnte eine Überprüfung der ADF in der Modellierung anzeigen, ob die beschriebenen Agenten

[7] siehe Abschnitte 7.1.3 und 5.3.6

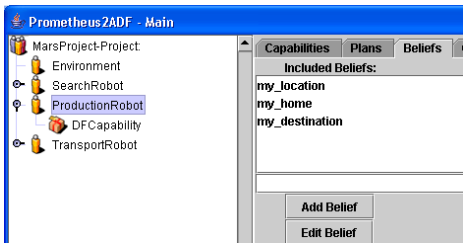


Abbildung 7.12: Bearbeitung der Agenten des Beispieles

ausführbar sind oder nicht. Ein Hilfesystem könnte sinnvolle Werte für zahlreiche Elemente vorschlagen. So könnten die benötigten Importanweisungen, die auswählbaren Sprachen und Ontologien vorgeschlagen werden. Auch für die möglichen Filter eines Planes könnten Vorschläge generiert werden.
An einigen Stellen können Ausdrücke in der Jadex eigenen Anfragesprache eingegeben werden. Eine Unterstützung bei der Einhaltung der Syntax wäre hilfreich.
Die Überführung der ADF in ein PDT–Projekt sind in dieser Phase unbefriedigend. Hauptsächlich werden die einzelnen Elemente des Systems in dem Projekt eingefügt. Ihr Anordnung zueinander ist aus den statischen Modellen nicht auslesbar. Durch eine Interpretation der jeweiligen Pläne eines Agenten könnten die Protokolle des Nachrichtenaustausches zwischen ihnen abgeleitet werden.

Aus Mangel an Ressourcen konnten diese Ansätze in dieser Arbeit nicht verfolgt werden. Dies mindert die Einsetzbarkeit des Prototypen. Die einzelnen ADF und einige, der in ihnen beschriebenen Eigenschaften, können zwar aus den Design–Modellen des Modellierungswerkzeuges gewonnen werden, aber der Benutzer muss noch eine Reihe von spezifischen Eigenschaften bestimmen. Hierbei könnte dem Benutzer weit mehr Hilfe angeboten werden, als implementiert ist. Da es sich hierbei um ein sog. *proof–of–concept* handelt, wurde für ihn auch keine Bewertung nach dem in Abschnitt 4.3 (siehe ebenfalls D) beschriebenen ISONORM–Fragebogen durchgeführt.
Zum Zeitpunkt dieser Arbeit beginnen weitere Diplomarbeiten an der Universität Hamburg, die sich mit der Modellierung der Agenten des Jadex Systems beschäftigen. Deren Ergebnisse mögen zu einer umfassenderen Werkzeugunterstützung führen, in der nicht nur, wie hier dargestellt, die statischen Eigenschaften der Agenten modelliert werden.

Kapitel 8

Zusammenfassung der Ergebnisse und Ausblick

8.1 Ergebnisse

In dieser Arbeit wurden verschiedene Entwicklungsansätze dahingehend untersucht, ob sie dazu geeignet sind, die Entwicklung von Anwendungen mit Hilfe des Jadex Systems zu unterstützen. Bei dieser Betrachtung wurde zunächst ein Überblick über die verfügbaren Methoden gegeben, um dann etablierte Ansätze zur Unterstützung von BDI–Architekturen eingehender zu betrachten.
Die bisherigen Arbeiten, die agentenorientierte Methoden betrachteten, wurden untersucht und ein Katalog von Kriterien entwickelt, der die Bedürfnisse einer Methode im Allgemeinen und des Jadex Systems im Besonderen erfasst. Anhand dieser Kriterien wurde aus verbreiteten Methoden, die die BDI–Architektur unterstützen, eine Auswahl getroffen. Ein Anwendungsbeispiel wurde modelliert, um einen Eindruck von deren Benutzbarkeit zu erlangen und so Aspekte zu bewerten, die sich einer einfachen, qualitativen Analyse entziehen.
Schlussendlich wurde die Prometheus–Methode ausgewählt. Die Übereinstimmung, der in dieser Methode modellierten Konzepte mit deren Benutzung in der Jadex Plattform, ist am größten. Durch die konsequente Benutzung von BDI–Konzepten und ein geeignetes Vorgehensmodell, werden insbesondere unerfahrene Entwickler in der Benutzung dieser Architektur und damit auch dem agentenorientierten Paradigma zur Softwareentwicklung, (an-)geleitet. Implementationsspezifische Eigenschaften einzelner Agenten werden am detailliertesten beschrieben.

Dies führte zu einer Untersuchung, inwieweit diese Informationen zwischen den Design–Artefakten der Prometheus–Methode und den detaillierten Beschreibungen der Agenten in Jadex übertragbar sind. Mit Hilfe eines Prototypen wurde der Umfang der automatischen Transformation gezeigt. Er erlaubt die Konvertierung von Projekt–Dateien eines Modellierungswerkzeuges in die XML–Beschreibungen einzelner Agenten der Jadex Plattform. Auch in die entgegengesetzte Richtung, kann aus einer einer Gruppe dieser XML–Dateien eine entsprechende Projekt–Datei generiert werden. Da die semantischen Unterschiede zwischen den Deklarationen der Agenten und den Modellen der Methode aber erheblich sind, ist ein Informationsverlust bei diesen Konvertierungen unvermeidlich. Es wurde aufgezeigt, welche Informationen

übertragbar sind, und welche systemspezifischen Details vom Anwender angegeben werden müssen. Hierzu wird die bequeme Editierung der ADF in einer grafischen Oberfläche ermöglicht.

In einer Beispielanwendung, die eine festen Teamorganisation von Agenten realisiert, wurde die Benutzung des Prototypen gezeigt und seine Anwendbarkeit bewertet. Es wurde dargestellt, dass durch den eben beschriebenen Informationsverlust der Aufwand für den Benutzer erheblich ist. Auch der Prozess zeigt einige Schwächen, die Ausdrucksstärke vieler der in Jadex benutzten Konzepte wird nicht berücksichtigt.

Alle in dieser Arbeit betrachteten Methoden beanspruchen grundsätzlich plattformunabhängig zu sein. Dies trifft nur bedingt zu. Beispielsweise konnten sich verschiedene Ausprägungen objektorientierter Methoden entwickeln, da ein allgemein anerkanntes Verständnis davon existierte, was unter der Objektorientierung an sich zu verstehen ist. Auch wenn nur eine bestimmte Architektur, in diesem Fall der BDI–Aufbau, betrachtet wird, so sind die Unterschiede in den verschiedenen Implementationen immer noch erheblich. In dieser Arbeit wurden andere Implementationen von BDI Systemen nicht betrachtet, aber die Differenzen, die die Übertragung von Informationen zwischen den Prometheusmodellen und dem Jadex System beschränkten, zeigen dies. In beiden Fällen werden BDI–Konzepte beschrieben, aber die Unterschiede sind umfangreich und lassen eine automatische Transformation kaum zu.

Die in dieser Arbeit angestellten Überlegungen, wie die Eignung einer Methode für eine bestimmte Agentenplattform zu bestimmen ist, wurde in ein allgemeines Schema zum Vergleich von agentenorientierten Methoden weiterentwickelt. Ein entsprechender Konferenzbeitrag (Sudeikat et al., 2004) wurde zum *Fifth International Workshop on AGENT-ORIENTED SOFTWARE ENGINEERING*[1] (AOSE-2004) zugelassen. Eine Bewertung nach diesem Schema basiert auf der Betrachtung der Übereinstimmung zwischen Methoden und Plattformen. Neben plattformunabhängigen Eigenschaften, die die Qualität von Methoden beeinflussen, werden plattformabhängige Kriterien klassifiziert. Für eine konkrete Untersuchung, wird die Identifikation der letztgenannten aus einer Kombination von Methode(n) und Plattform(en) angeleitet. Die in dieser Diplomarbeit gefundenen Kriterien und Ergebnisse dienen als Beispiel und leiten zukünftige Betrachtungen an.

8.2 Ausblick

Bei der Betrachtung der einzelnen Methoden wurden unter anderem alle deren Arbeitsschritte betrachtet. Da aber auch eine mögliche Werkzeugunterstützung mit in Betracht gezogen wurde, konzentrierte sich die hier beschriebene Untersuchung, und damit auch der Kriterienkatalog, auf die Modelle des Design. Interessante Ergebnisse sind aus einer Untersuchung der Methoden mittels des von Yu und Cysneiros (2002) vorgeschlagenen Challenge Exemplar (kurz beschrieben in Abschnitt 6.5) zu erwarten. In einer derartigen Gegenüberstellung würden insbesondere die Möglichkeiten der Modellierung von Zielen und der Abhängigkeiten der Agenten untereinander analysierbar sein. Die in dieser Arbeit betrachteten Beispiele sind zu überschaubar, um die Möglichkeiten der Methoden hierbei aufzuzeigen.

[1] http://www.jamesodell.com/aose2004

Für die Entwicklung von Agentensystemen sind bisher in nur wenigen Projekten Erfahrungswerte gesammelt worden. Diese mangelnde Erfahrung ist mit Sicherheit auch der Grund dafür, dass das Management eines agentenorientierten Softwareprojektes bisher vernachlässigt wurde. Die hier betrachteten Methoden befinden sich daher auch in einem frühen Stadium ihrer Entwicklung, ähnlich der Evolution ihrer objektorientierten Vorgänger definieren sie zunächst eigene Modelle und Notationen, bevor die Vorgehensweisen vervollkommnet werden können. Die betrachten Methoden versuchten meist anerkannte Prinzipien und Techniken der Softwareentwicklung auf Agentensysteme umzudeuten und anwendbar zu machen. Es stellt sich die Frage, inwieweit weitere Techniken adaptierbar sind. Mit zunehmender Verbreitung sind hier aufschlussreiche Ergebnisse zu erwarten. Beispielsweise bei der Adaption von Aufwandsabschätzungen, Qualitätssicherung, etc.

Bemerkenswerterweise entwickelte sich der verbreitete Rational Unified Process aus einer Zusammenführung verschiedener Methoden. In dieser Arbeit wurde eine Auswahl getroffen, interessante Ansätze könnten versuchen, eine Art 'Unified Process' für Agentensysteme zu extrahieren. Größte Schwierigkeit hierbei ist die oben angesprochene Plattformabhängigkeit.
Verfechter könnten einwenden, dass diese Unterschiede nur während des Designs auftreten und daher ein gemeinsames Vorgehensmodell entwickelt werden kann, der dann ein plattformabhängiger Designarbeitsschritt angeschlossen wird. Aber dieser Ansatz ist problematisch. Methoden sollten die BDI–Konzepte ganzheitlich unterstützen (deren Unterstützung in lediglich einem Arbeitsschritt war ein Kritikpunkt für MaSE). Beispielsweise ist eine Analyse eines Systems ohne eine genaue Modellierung der verwendbaren Ziele (und deren Eigenschaften) nicht zweckdienlich.

Es zeichnen sich folglich zwei Richtungen für zukünftige Entwicklungen ab. Langfristig könnten die Möglichkeiten der Zusammenführung von Methoden zu einem umfassenden Prozess weiter untersucht werden, kurzfristig werden sich zukünftige Anstrengungen auf Anpassungen/Erweiterungen bestehender Methoden für konkrete Plattformen konzentrieren, um die Anwendungsentwicklung zu unterstützen. Das *FIPA Methodology Technical Committee*[2] verfolgt derzeit den ersten Ansatz. Ziel seiner Arbeit ist es, einzelne Teile der verschiedenen Methoden, die FIPA–konform sind, zu extrahieren, um Entwicklern eine Möglichkeit der Zusammenstellung eines Prozesses zu geben. Dies ist ein Ansatz, aber unabhängig von den hier betrachteten Problematik der verschiedenen BDI–Implementationen. Die folgenden Anregungen sind denkbar, um eine weitere Anpassung an die Jadex–Plattform zu erreichen.

Wie Abschnitt 7.1.3 zeigte, unterstützt die gefundene Methode viele Jadex eigene Eigenschaften nicht. Insbesondere Erweiterungen der Methode, um das Meta–Level–Reasoning darzustellen und die Einführung von UML–Diagrammen, um den Ablauf innerhalb der Pläne darzustellen, sollten untersucht werden. Dies würde die Ausdruckskraft der Methode für diese spezielle Plattform erhöhen.
Während der Fertigstellung dieser Arbeit wurde damit begonnen Jadex um ein weiteres Werkzeug zu erweitern. Der *Agent Society Configuration Manager and Loader* (ASCML) wird die Definition, Organisation und Konfiguration von Agentengesellschaften (sog. *Societies*) ermöglichen. In XML–Dateien werden die einzelnen ADF

[2] http://www.fipa.org/activities/methodology.html

referenziert und Szenarien beschrieben, die es erlauben mehrere Agenten, als gemeinsame Anwendung zu starten. Die Möglichkeiten der Ableitung dieser Beschreibungen von Gesellschaften aus den Modellen von Methoden sind zu untersuchen. Es ist zu erwarten, dass diese abstraktere Sicht auf ein System akkurater abgebildet werden kann, als die detaillierten ADF.

Interessant sind die Ansätze, die die sog. frühen Anforderungen an ein System erheben. Alle hier betrachteten Methoden zeigten Schwächen in der Modellierung der Ziele. In Jadex sind die Ziele, durch eine Reihe von Eigenschaften, sehr genau beschreibbar. So gibt es beispielsweise verschiedene Arten von Zielen und Eigenschaften des Meta–Level–Reasoning sind bestimmbar, um zu beeinflussen, wie die einzelnen Ziele von Agenten ausgewählt werden. Die Methoden beschränken sich meist darauf die Ziele eines Systems in einer Hierarchie darzustellen und Agenten zuzuordnen. Da die Ziele ein zentrales Element der BDI–Architektur sind und ihre Definition maßgeblichen Einfluss auf die Arbeitsweise der Agenten hat, scheint es sinnvoll, zu untersuchen, wie diese anschaulicher modelliert werden können. Um diese Eigenschaften beschreibbar zu machen, können Ansätze des Requirements Engineering Anregungen geben. Wie solche Ansätze für die Modellierung von Agenten eingesetzt werden können, wurde an der Tropos–Methode gezeigt. Ein weiteres Framework, das die Ziele eines Softwaresystems beschreibt, ist der KAOS Ansatz (beschrieben von van Lamsweerde, 2003). Hier werden die Ziele eines Systems ebenfalls in eine Hierarchie eingeordnet, durch Vor- und Nachbedingungen beschrieben und so weit in Teilziele aufgegliedert, bis atomare Aktionen beschrieben sind, anhand derer die Zielerfüllung eines Systems bewiesen werden kann.
Diese Ansätze können nur Anregungen geben, eine Anpassung bzw. Entwicklung einer geeigneten Modellierung, die den Eigenschaften von Jadex Rechnung trägt, ist eine umfassende Aufgabe, der hier nur angedacht werden kann.

In einem Projekt der Universität Ulm wird die Anwendbarkeit des *Extreme Programming* untersucht. Knublauch (2002) beschreibt Erfahrungen in einem entwickelten Agentensystem und ein eigens entwickeltes Modellierungswerkzeug. In diesem Werkzeug wird ein (Agenten–)Systems als eine Sammlung interagierender Prozesse modelliert. Leider wird eine BDI–Architektur der Agenten nicht in dem Maße berücksichtigt, wie es für eine so umfassende Implementation wie das Jadex System nötig wäre. Ansätze der *agilen* Software–Entwicklung, wie das Extreme Programming, stellen einige Anforderungen an die Entwicklung und deren Werkzeuge, vor allem die Testbarkeit der entwickelten Systeme ist unabdingbar. Da die Agenten als autonome Prozesse verstanden und objektorientiert programmiert werden, können hierzu traditionelle, objektorientierte Tests benutzt werden. Das Testen und Debuggen von BDI-Systemen allerdings ist ein aktives Forschungsfeld. Die bisher vorgeschlagenen Ansätze konzentrieren sich auf die Überprüfung der zwischen den Agenten stattfindenden Kommunikation (beschrieben von: Liedekerke und Avouris, 1995; Poutakidis, Padgham und Winikoff, 2002; 2003).
Trotzdem erscheint eine Untersuchung, inwieweit eine Unterstützung des Extreme Programming für Agentensysteme der BDI-Architektur möglich wäre, vielversprechend. So ist zu erwarten, dass das dort oft verwendete *Refactoring* durch die Autonomie der Agenten positiv unterstützt wird. Die Kriterien, die in dieser Arbeit aufgestellt wurden, wären für diese Untersuchung wenig hilfreich, da die Eigenschaften dieses agilen Ansatzes zur Entwicklung von anderen Voraussetzungen und Problemstellungen ausgehen. Eine solche Untersuchung würde eng verbunden

sein mit der Weiterentwicklung der Möglichkeiten, Agenten automatisiert testen zu können. Ein Schritt in diese Richtung ist die *JADE Test Suite* (Cortese, Caire und Bochicchio, 2004), die sich an bekannten Testtreibern für Komponententests in objektorientierten Sprachen wie *JUnit*[3] orientiert, um automatisierte Tests für einzelne Behaviours der JADE–Agenten, durchführen zu können.

In Abschnitt 5.2.1 wurde ein Überblick über Ansätze gegeben, die mit Design Pattern die Anwendungsentwicklung unterstützen wollen. Die Prämisse, dass bei der Softwareentwicklung oft wiederkehrende Probleme gelöst werden müssen und so ein Katalog von generischen, bewährten Lösungen für diese Probleme hilfreich ist, gilt sicherlich auch für Agentensysteme. Außerdem würde eine solche Sammlung von nachvollziehbaren Konstellationen von Agenten zukünftigen Anwendern die Einarbeitung in das System erleichtern. Studenten könnten von anschaulichen Beispielen für ein agentenorientiertes Design profitieren.
Wie in dem genannten Abschnitt gezeigt, müssten diese Pattern aber spezifisch für die verwendete Plattform sein. Eine Definition solcher Pattern müsste sich wohl an der Definition der sog. *social pattern* (kurz beschrieben in den Abschnitten 5.2.2 und 5.3.4) der Tropos–Methode orientieren, um den mentalistischen Konzepten Rechnung zu tragen.

[3] www.junit.org

Anhang A

Schematische Darstellung der ADF

Die folgenden beiden Abbildungen stellen den schematischen Aufbau der ADF zur Beschreibung eines Agenten und einer Capability dar.
In diesen Darstellungen wird nur die Struktur der aufeinander folgenden Elemente der XML-Dateien dargestellt. Die Bedeutung der einzelnen Elemente, sowie etwaiger Attribute wird in Abschnitt 7.1.1 beschrieben. Diese Übersichten wurden mittels des kommerziellen Werkzeuges XMLSpy™) aus der entsprechenden XML–Schema Definition[1] generiert.

[1] Namespace–Bezeichner und online verfügbar: http://jadex.sourceforge.net/jadex.xsd

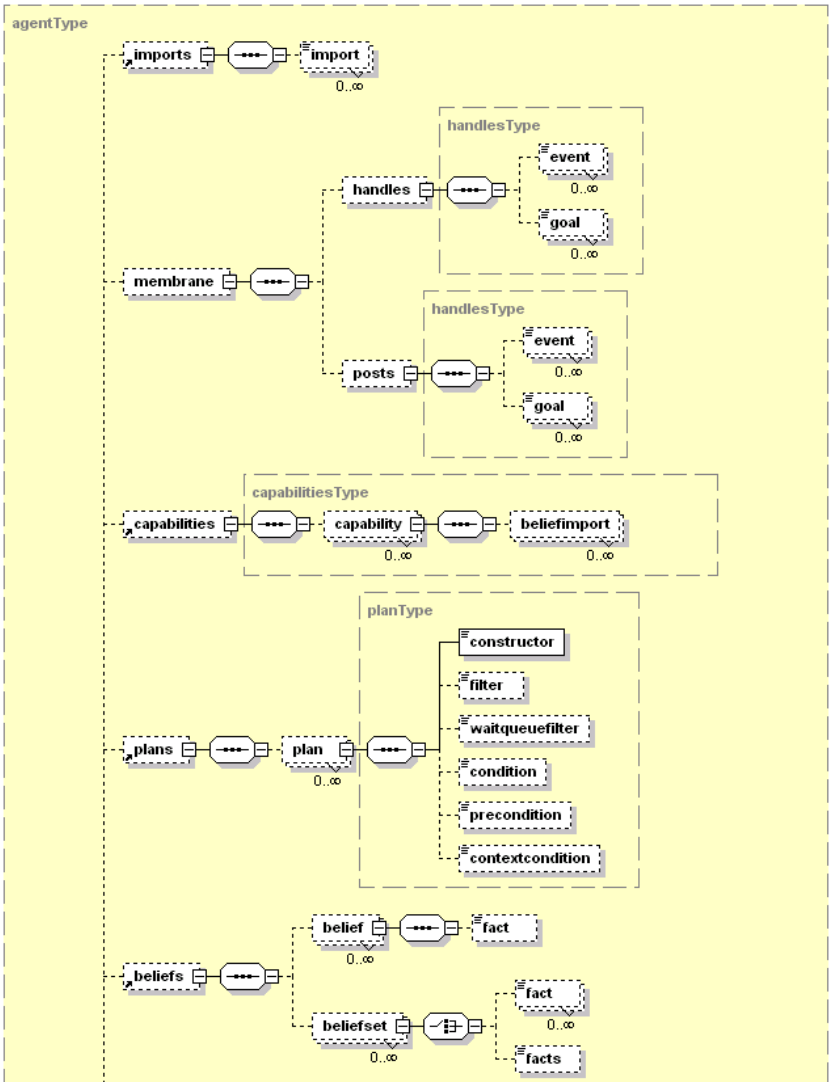


Abbildung A.1: Grafische Darstellung des ADF Schema - Teil 1

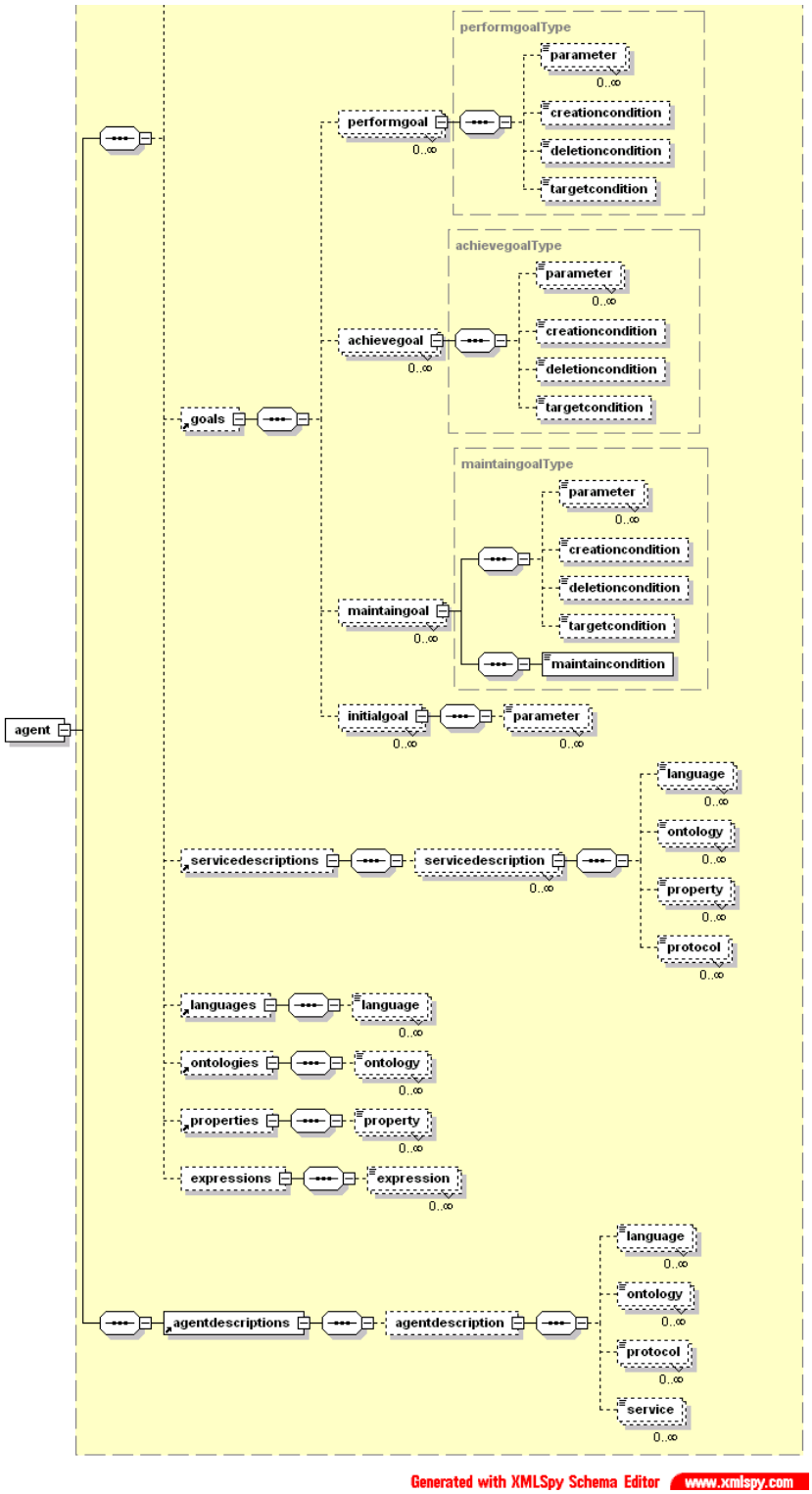


Abbildung A.2: Grafische Darstellung des ADF Schema - Teil 2

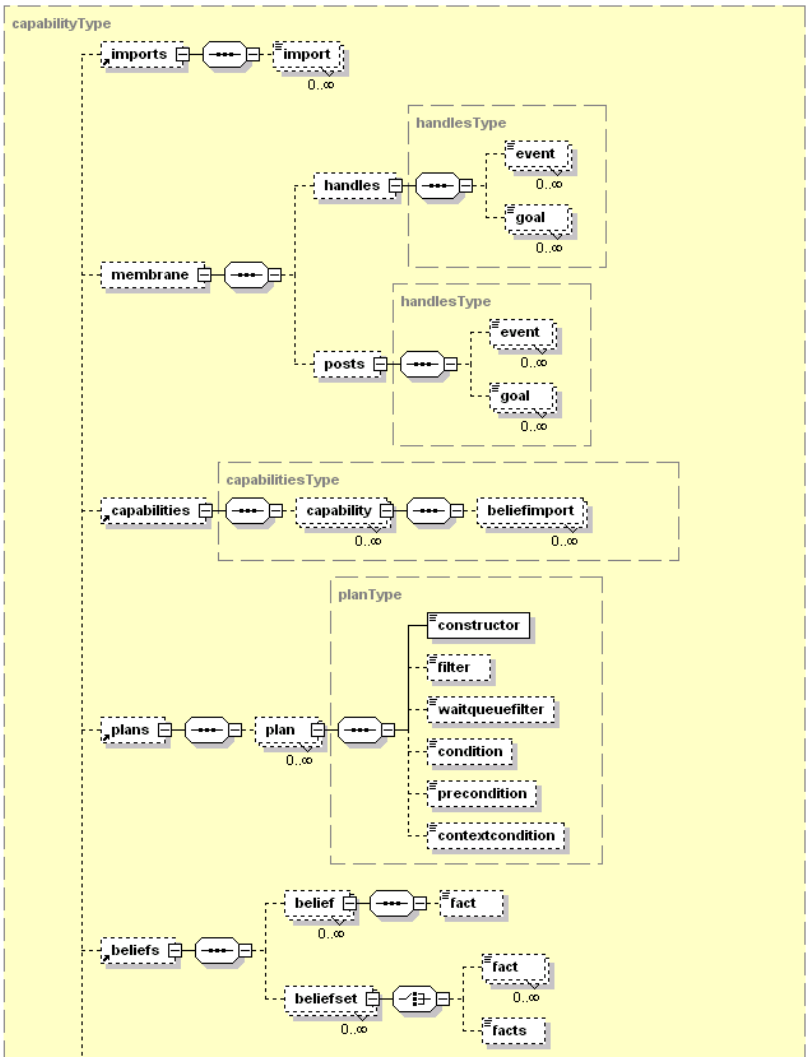


Abbildung A.3: Grafische Darstellung eines Capability ADF - Teil 1

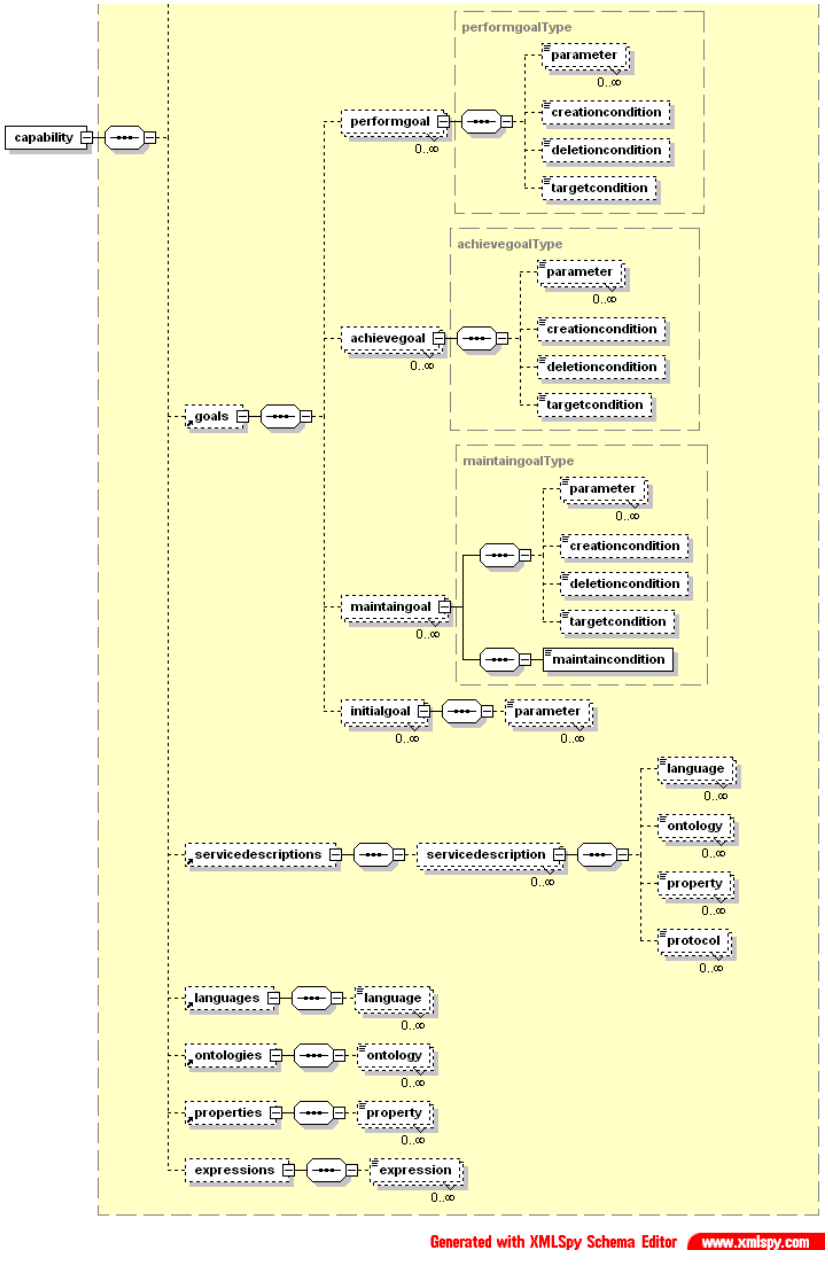


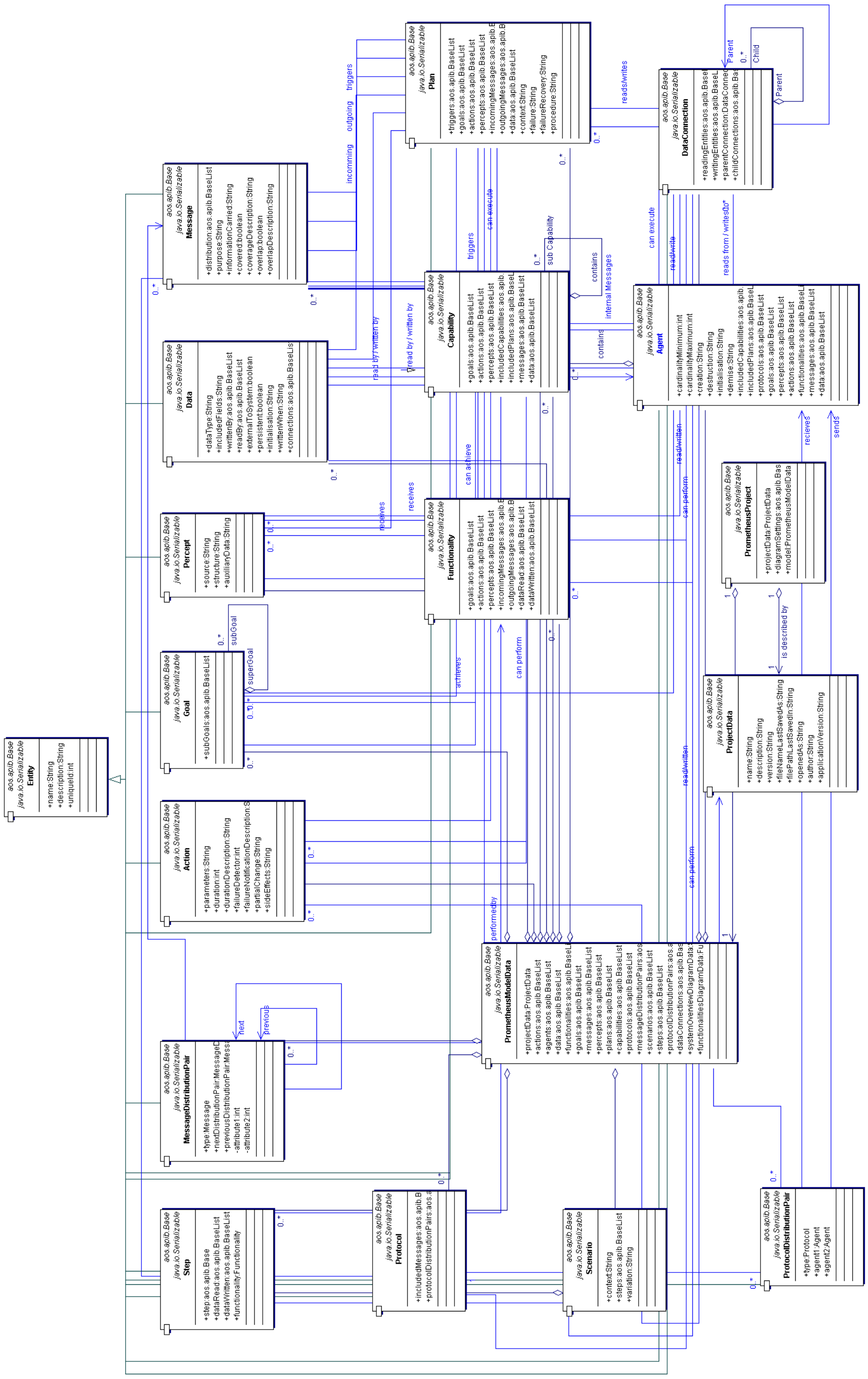
Abbildung A.4: Grafische Darstellung eines Capability ADF - Teil 2

Anhang B

Metamodel eines PDT–Projektes

Abbildung B.1 zeigt die Struktur von Beschreibungen einzelner Agentensysteme (gespeichert in sog. Projekten), wie sie vom Prometheus Design Tool (PDT) verwendet werden. Dieses Modell wurde aus den Klassen gewonnen, die vom PDT zur Speicherung von Projekten, mittels des JACOB Object Modeller (beschrieben in 7.2.1), verwendet werden. Um das verwendete Meta-Modell darzustellen, wurde die tatsächlich verwendete Klassenstruktur auf die Klassen reduziert, die agentenorientierte Konzepte darstellen. In dem tatsächlich verwendeten Modell sind zahlreiche Klassen vorhanden, die die dargestellten Diagramme repräsentieren.

Die Klasse *PrometheusProject* ist in der rechten unteren Seite des Klassendiagrammes dargestellt. Sie hat Aggregationen zu zwei weiteren Klassen. *ProjectData* enthält zusätzliche Informationen über die das Projekt, *PrometheusModelData* enthält alle agentenorientierten Elemente, die modelliert werden.
Alle diese Elemente sind Unterklassen der Klasse *Entity* (oben in der Mitte dargestellt). Die Klassen *Agent* und *Capability* beschreiben die entsprechenden Elemente eines Agentensystems. Diese haben Referenzen auf die weiteren Elemente.
Die Verbindung zu einer Beliefbase (modelliert durch die Klasse *Data*) werden durch die Klasse *DataConnection* beschrieben.
Der Nachrichtenaustausch im System wird durch die Klasse *ProtocolDistributionPair* dargestellt. Es enthält eine Assoziation zu der Klasse *Protocol*, die wiederum die einzelnen Nachrichten referenziert.
Bemerkenswerterweise werden die einzelnen Nachrichten, die einen Plan empfangen bzw. versenden kann in der Klasse MessagedistributionPair beschrieben (die wiederum die einzelnen Nachrichten referenziert). Die Klasse *Scenario* beschreibt die Anwendungsfälle für das zu entwickelnde System. Die Klasse *Step* beschreibt die einzelnen Aktionen, die in einem so dargestellten Anwendungsfall ausgeführt werden können.
Die Klassen *Action* und *Percept* beschreiben die modellierten Interaktionen des Systems mit der Umwelt. In der Klasse *Funcionality* werden die Aktionen beschrieben, die ein Agent ausführen kann.
In den Klassen *Goal* und *Plan* werden die entsprechende Konzepte der BDI-Architektur beschrieben.


Entity
Message
Data
Percept
Goal
Action
MessageDistributionPair
Step
Protocol
Scenario
PrometheusModelData
Plan
Capability
Functionality
Agent
DataConnection
PrometheusProject
ProjectData
ProtocolDistributionPair

Anhang C

Package–Struktur des Prototypen

Abbildung C.1 zeigt die Abhängigkeiten der Packages des in Abschnitt 7.2 beschriebenen Prototypen. Er besthet aus den folgenden Packages:

`jadex.tools.pdt2adf.gui` stellt die Klassen zur Darstellung der Benutzungsschnittstelle bereit. In einem Unterverzeichnis (`jadex.tools.pdt2adf.gui.images`) sind die verwendeten Bilder zur Darstellung zu finden.

`jadex.tools.pdt2adf.helpers` Eine Sammlung von Hilfsklassen, die für weitere Aufgaben, wie Einlesen und Speichern der PDT-Projekte/XML–Dateien und Konvertierungen, verantwortlich sind. Sie kapseln hierzu die Verwendung der weiter unten aufgeführen Werkzeuge. In einem Unterverzeichnis (`jadex.tools.pdt2adf.helpers.templates`) sind die von der Template Engine (beschrieben in Abschnitt 7.2.1) verwendeten Templates abgelegt.

`jadex.tools.pdt2adf.model` eine objektorientierten Repräsentation der Struktur eines ADF. Sie hält lediglich die Zeichenketten und validiert die Werte nicht.

`jadex.tools.pdt2adf.wrapper` In diesem Package ist ein (Klassen–)Adapter (wie beschrieben von Gamma et. al, 1997), für die Jadex eigene Repräsentation der ADF (`jadex.model`) enthalten. In dem enthaltenen Package `jadex.tools.pdt2adf.wrapper.interfacessind` Interfaces enthalten, die die möglichen Zugriffe auf die Repräsentationen der ADF definieren. Die Dialoge der Benutzungsschnittstelle und die Templates zur Generierung der XML–Dateien benutzen diese.

Des Weiteren werden verwendet:

`jadex.model` Die interne Repräsentation eines ADF des Jadex Systems. Beim Instanziieren dieser Strukturen werden die eingelesenen Werte explizit überprüft.

`jadex.tools.loader` Ein Werkzeug des Jadex Systems, mit dessen Hilfe XML–Dateien eingelesen und in die obige interne Repräsentation übersetzt werden können.

`aos.apib` Die Klassen des JACOB Object Modeller (beschrieben in Abschnit 7.2.1). Sie werden verwendet, um Projekt-Dateien des PDT einzulesen, und zu speichern.

`au.edu.rmit.cs.prometheus.datamodel.jacob` Diese Klassen wurden aus den Dictionary–Dateien für den JACOB Object Modeller. Sie werden von PDT benutzt, um die Elemente eines Projektes zu beschreiben.

`org.apache.velocity` Die Velocity Template Engine. Sie wird zur Generierung von XML–Dateien verwendet.

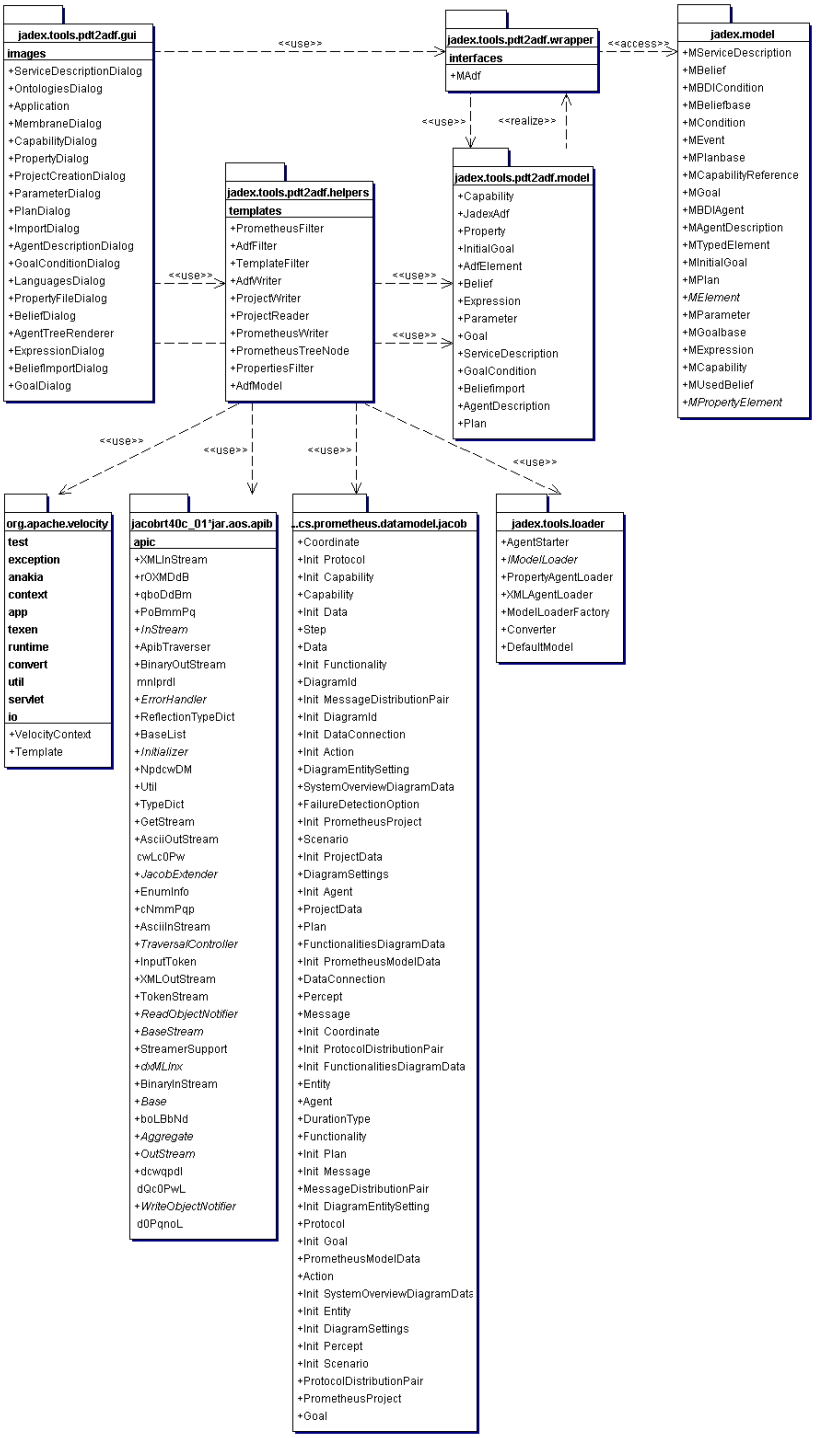


Abbildung C.1: Die Packages des Prototypen

Anhang D

Fragebögen der ISO NORM 9241/10 - Bewertungen

In dieser Arbeit wurden zwei Werkzeuge zur Modellierung von Agentensystemen eingehender betrachtet. Die folgenden Seiten enthalten die Fragen, wie sie im ISONORM–Fragebogen von Prümper und Anft (1993) aufgestellt wurden.
Der eigentliche Fragebogen ist für eine statistische Auswertung mit einer großen Gruppe von Befragten gedacht. Da die Mittel hierzu fehlten, beschreiben die folgenden Seiten den Eindruck, den der Autor dieser Arbeit von den Werkzeugen gewonnen hat, und wie sie insbesondere in Kapitel 6 beschrieben wurden.

Der Fragebogen will sieben Gestaltungsgrundsätze, wie sie in der Software–Ergonomie–Norm DIN EN ISO 9241 Teil 10 (DIN, 1996) beschrieben werden, überprüfen. Jeder dieser Grundsätze wird in fünf Einzelfragen abgefragt. Für die Antworten wird jeweils ein siebenstufiges Bewertungsschema verwendet, von sehr negativ (– – –) bis sehr positiv (+++).
In seine originalen Form beinhaltet er weitere Fragen, in denen die befragten Anwender ihre Kenntnisse der Software beschreiben sollen. Diese Fragen werden hier nicht erwähnt.

D.1 Bewertung des AgentTool

Name : AgentTool
Versionsnummer: 2.0 Beta
Hersteller: Kansas State Universtiy [www.cis.ksu.edu/ sdeloach/ai/agentool.htm]

Aufgabenangemessenheit

Unterstützt die Software die Erledigung Ihrer Arbeitsaufgaben, ohne Sie als Benutzer unnötig zu belasten?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
ist kompliziert zu bedienen.						X		ist unkompliziert zu bedienen.
bietet nicht alle Funktionen, um die anfallenden Aufgaben effizient zu bewältigen.							X	bietet alle Funktionen, um die anfallenden Aufgaben effizient zu bewältigen.
bietet schlechte Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.		X						bietet gute Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.
erfordert überflüssige Eingaben.							X	erfordert keine überflüssigen Eingaben.
ist schlecht auf die Anforderungen der Arbeit zugeschnitten.						X		ist gut auf die Anforderungen der Arbeit zugeschnitten.

Tabelle D.1: Bewertung der Aufgabenangemessenheit

Selbstbeschreibungsfähigkeit

Gibt Ihnen die Software genügend Erläuterungen und ist sie in ausreichendem Maße verständlich?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
bietet einen schlechten Überblick über ihr Funktionsangebot.						X		bietet einen guten Überblick über ihr Funktionsangebot
verwendet schlecht verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.						X		verwendet gut verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.
liefert in unzureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.				X				liefert in zureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.
bietet auf Verlangen keine situationsspezifischen Erklärungen, die konkret weiterhelfen.		X						bietet auf Verlangen situationsspezifische Erklärungen, die konkret weiterhelfen.
bietet von sich aus keine situationsspezifischen Erklärungen, die konkret weiterhelfen.		X						bietet von sich aus situationsspezifische Erklärungen, die konkret weiterhelfen.

Tabelle D.2: Bewertung der Selbstbeschreibungsfaehigkeit

Steuerbarkeit

Können Sie als Benutzer die Art und Weise, wie Sie mit der Software arbeiten, beeinflussen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
bietet keine Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.							X	bietet die Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.
erzwingt eine unnötig starre Einhaltung von Bearbeitungsschritten.							X	erzwingt keine unnötig starre Einhaltung von Bearbeitungsschritten.
ermöglicht keinen leichten Wechsel zwischen einzelnen Menüs oder Masken.						X		ermöglicht einen leichten Wechsel zwischen einzelnen Menüs oder Masken.
ist so gestaltet, daß der Benutzer nicht beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.						X		ist so gestaltet, daß der Benutzer beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.
erzwingt unnötige Unterbrechungen der Arbeit.							X	erzwingt keine unnötigen Unterbrechungen der Arbeit.

Tabelle D.3: Bewertung der Steuerbarkeit

Erwartungskonformität

Kommt die Software durch eine einheitliche und verständliche Gestaltung Ihren Erwartungen und Gewohnheiten entgegen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
erschwert die Orientierung, durch eine uneinheitliche Gestaltung.						X		erleichtert die Orientierung, durch eine einheitliche Gestaltung.
läßt einen im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.							X	läßt einen nicht im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.
informiert in unzureichendem Maße über das, was sie gerade macht.						X		informiert in ausreichendem Maße über das, was sie gerade macht.
reagiert mit schwer vorhersehbaren Bearbeitungszeiten.							X	reagiert mit gut vorhersehbaren Bearbeitungszeiten.
läßt sich nicht durchgehend nach einem einheitlichen Prinzip bedienen.							X	läßt sich durchgehend nach einem einheitlichen Prinzip bedienen.

Tabelle D.4: Bewertung der Erwartungskonformität

Fehlertoleranz

Bietet Ihnen die Software die Möglichkeit, trotz fehlerhafter Eingaben das beabsichtigte Arbeitsergebnis ohne oder mit geringem Korrekturaufwand zu erreichen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
ist so gestaltet, daß kleine Fehler schwerwiegende Folgen haben können.						X		ist so gestaltet, daß kleine Fehler keine schwerwiegende Folgen haben können.
informiert zu spät über fehlerhafte Eingaben.							X	informiert sofort über fehlerhafte Eingaben.
liefert schlecht verständliche Fehlermeldungen.						X		liefert gut verständliche Fehlermeldungen.
erfordert bei Fehlern im großen und ganzen einen hohen Korrekturaufwand.				X				erfordert bei Fehlern im großen und ganzen einen geringen Korrekturaufwand.
gibt keine konkreten Hinweise zur Fehlerbehebung.			X					gibt konkrete Hinweise zur Fehlerbehebung.

Tabelle D.5: Bewertung der Fehlertoleranz

Individualisierbarkeit

Können Sie als Benutzer die Software ohne großen Aufwand auf Ihre individuellen Bedürfnisse und Anforderungen anpassen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
läßt sich von dem Benutzer schwer erweitern, wenn für ihn neue Aufgaben entstehen.					X			läßt sich von dem Benutzer leicht erweitern, wenn für ihn neue Aufgaben entstehen.
läßt sich von dem Benutzer schlecht an seine persönliche, individuelle Art der Arbeitserledigung anpassen.				X				läßt sich von dem Benutzer gut an seine persönliche, individuelle Art der Arbeitserledigung anpassen.
eignet sich für Anfänger und Experten nicht gleichermaßen, weil der Benutzer sie nur schwer an seinen Kenntnisstand anpassen kann.				X				eignet sich für Anfänger und Experten gleichermaßen, weil der Benutzer sie leicht an seinen Kenntnisstand anpassen kann.
läßt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer schlecht für unterschiedliche Aufgaben passend einrichten.				X				läßt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer gut für unterschiedliche Aufgaben passend einrichten.
ist so gestaltet, daß der Benutzer die Bildschirmdarstellung schlecht an seine individuellen Bedürfnisse anpassen kann.		X						ist so gestaltet, daß der Benutzer die Bildschirmdarstellung gut an seine individuellen Bedürfnisse anpassen kann.

Tabelle D.6: Bewertung der Individualisierbarkeit

Lernförderlichkeit

Ist die Software so gestaltet, daß Sie sich ohne großen Aufwand in sie einarbeiten konnten und bietet sie auch dann Unterstützung, wenn Sie neue Funktionen lernen möchten?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
erfordert viel Zeit zum Erlernen.						X		erfordert wenig Zeit zum Erlernen.
ermutigt nicht dazu, auch neue Funktionen auszuprobieren.					X			ermutigt dazu, auch neue Funktionen auszuprobieren.
erfordert, daß man sich viele Details merken muß.					X			erfordert nicht, daß man sich viele Details merken muß.
ist so gestaltet, daß sich einmal Gelerntes schlecht einprägt.						X		ist so gestaltet, daß sich einmal Gelerntes gut einprägt.
ist schlecht ohne fremde Hilfe oder Handbuch erlernbar.							X	ist gut ohne fremde Hilfe oder Handbuch erlernbar.

Tabelle D.7: Bewertung der Lernförderlichkeit

D.2 Bewertung des Prometheus Desing Tool

Name : Prometheus Design Tool (PDT)
Versionsnummer: 1.1
Hersteller: Royal Melbourne Institute of Technology [http://www.cs.rmit.edu.au/agents/pdt/]

Aufgabenangemessenheit

Unterstützt die Software die Erledigung Ihrer Arbeitsaufgaben, ohne Sie als Benutzer unnötig zu belasten?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
ist kompliziert zu bedienen.						X		ist unkompliziert zu bedienen.
bietet nicht alle Funktionen, um die anfallenden Aufgaben effizient zu bewältigen.						X		bietet alle Funktionen, um die anfallenden Aufgaben effizient zu bewältigen.
bietet schlechte Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.		X						bietet gute Möglichkeiten, sich häufig wiederholende Bearbeitungsvorgänge zu automatisieren.
erfordert überflüssige Eingaben.							X	erfordert keine überflüssigen Eingaben.
ist schlecht auf die Anforderungen der Arbeit zugeschnitten.						X		ist gut auf die Anforderungen der Arbeit zugeschnitten.

Tabelle D.8: Bewertung der Aufgabenangemessenheit

Selbstbeschreibungsfähigkeit

Gibt Ihnen die Software genügend Erläuterungen und ist sie in ausreichendem Maße verständlich?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
bietet einen schlechten Überblick über ihr Funktionsangebot.					X			bietet einen guten Überblick über ihr Funktionsangebot
verwendet schlecht verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.							X	verwendet gut verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.
liefert in unzureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.					X			liefert in zureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.
bietet auf Verlangen keine situationsspezifischen Erklärungen, die konkret weiterhelfen.		X						bietet auf Verlangen situationsspezifische Erklärungen, die konkret weiterhelfen.
bietet von sich aus keine situationsspezifischen Erklärungen, die konkret weiterhelfen.		X						bietet von sich aus situationsspezifische Erklärungen, die konkret weiterhelfen.

Tabelle D.9: Bewertung der Selbstbeschreibungsfaehigkeit

Steuerbarkeit

Können Sie als Benutzer die Art und Weise, wie Sie mit der Software arbeiten, beeinflussen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
bietet keine Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.							X	bietet die Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.
erzwingt eine unnötig starre Einhaltung von Bearbeitungsschritten.					X			erzwingt keine unnötig starre Einhaltung von Bearbeitungsschritten.
ermöglicht keinen leichten Wechsel zwischen einzelnen Menüs oder Masken.						X		ermöglicht einen leichten Wechsel zwischen einzelnen Menüs oder Masken.
ist so gestaltet, daß der Benutzer nicht beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.							X	ist so gestaltet, daß der Benutzer beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.
erzwingt unnötige Unterbrechungen der Arbeit.							X	erzwingt keine unnötigen Unterbrechungen der Arbeit.

Tabelle D.10: Bewertung der Steuerbarkeit

Erwartungskonformität

Kommt die Software durch eine einheitliche und verständliche Gestaltung Ihren Erwartungen und Gewohnheiten entgegen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
erschwert die Orientierung, durch eine uneinheitliche Gestaltung.							X	erschwert die Orientierung, durch eine einheitliche Gestaltung.
läßt einen im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.					X			läßt einen nicht im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.
informiert in unzureichendem Maße über das, was sie gerade macht.						X		informiert in ausreichendem Maße über das, was sie gerade macht.
reagiert mit schwer vorhersehbaren Bearbeitungszeiten.							X	reagiert mit gut vorhersehbaren Bearbeitungszeiten.
läßt sich nicht durchgehend nach einem einheitlichen Prinzip bedienen.							X	läßt sich durchgehend nach einem einheitlichen Prinzip bedienen.

Tabelle D.11: Bewertung der Erwartungskonformität

Fehlertoleranz

Bietet Ihnen die Software die Möglichkeit, trotz fehlerhafter Eingaben das beabsichtigte Arbeitsergebnis ohne oder mit geringem Korrekturaufwand zu erreichen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
ist so gestaltet, daß kleine Fehler schwerwiegende Folgen haben können.					X			ist so gestaltet, daß kleine Fehler keine schwerwiegende Folgen haben können.
informiert zu spät über fehlerhafte Eingaben.						X		informiert sofort über fehlerhafte Eingaben.
liefert schlecht verständliche Fehlermeldungen.			X					liefert gut verständliche Fehlermeldungen.
erfordert bei Fehlern im großen und ganzen einen hohen Korrekturaufwand.					X			erfordert bei Fehlern im großen und ganzen einen geringen Korrekturaufwand.
gibt keine konkreten Hinweise zur Fehlerbehebung.			X					gibt konkrete Hinweise zur Fehlerbehebung.

Tabelle D.12: Bewertung der Fehlertoleranz

Individualisierbarkeit

Können Sie als Benutzer die Software ohne großen Aufwand auf Ihre individuellen Bedürfnisse und Anforderungen anpassen?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
läßt sich von dem Benutzer schwer erweitern, wenn für ihn neue Aufgaben entstehen.					X			läßt sich von dem Benutzer leicht erweitern, wenn für ihn neue Aufgaben entstehen.
läßt sich von dem Benutzer schlecht an seine persönliche, individuelle Art der Arbeitserledigung anpassen.				X				läßt sich von dem Benutzer gut an seine persönliche, individuelle Art der Arbeitserledigung anpassen.
eignet sich für Anfänger und Experten nicht gleichermaßen, weil der Benutzer sie nur schwer an seinen Kenntnisstand anpassen kann.				X				eignet sich für Anfänger und Experten gleichermaßen, weil der Benutzer sie leicht an seinen Kenntnisstand anpassen kann.
läßt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer schlecht für unterschiedliche Aufgaben passend einrichten.				X				läßt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer gut für unterschiedliche Aufgaben passend einrichten.
ist so gestaltet, daß der Benutzer die Bildschirmdarstellung schlecht an seine individuellen Bedürfnisse anpassen kann.					X			ist so gestaltet, daß der Benutzer die Bildschirmdarstellung gut an seine individuellen Bedürfnisse anpassen kann.

Tabelle D.13: Bewertung der Individualisierbarkeit

Lernförderlichkeit

Ist die Software so gestaltet, daß Sie sich ohne großen Aufwand in sie einarbeiten konnten und bietet sie auch dann Unterstützung, wenn Sie neue Funktionen lernen möchten?

Die Software...	− − −	− −	−	−/+	+	++	+++	Die Software...
erfordert viel Zeit zum Erlernen.						X		erfordert wenig Zeit zum Erlernen.
ermutigt nicht dazu, auch neue Funktionen auszuprobieren.						X		ermutigt dazu, auch neue Funktionen auszuprobieren.
erfordert, daß man sich viele Details merken muß.						X		erfordert nicht, daß man sich viele Details merken muß.
ist so gestaltet, daß sich einmal Gelerntes schlecht einprägt.						X		ist so gestaltet, daß sich einmal Gelerntes gut einprägt.
ist schlecht ohne fremde Hilfe oder Handbuch erlernbar.							X	ist gut ohne fremde Hilfe oder Handbuch erlernbar.

Tabelle D.14: Bewertung der lernförderlichkeit

Anhang E

Glossar

Agent-based Unified Modeling Language (AUML): Erweiterung zu UML um agentenspezifische Konzepte ausdrücken zu können. Siehe: www.auml.org

Agent-Oriented Development (AOD): versteht sich als Erweiterung vom *Object-Oriented Development* (OOD). Der Begriff *Developement* wird gelegentlich als "Programmieren" an sich verstanden, meist aber als der gesamtheitlicher Entwicklungsprozess, der das Programmieren mit einschließt.

Agent-Oriented Software Engineering (AOSE): beschreibt die Entwicklung von agentenbasierten Softwaresystemen, schließt aber (im Gegensatz zu Agent-Oriented Development) auch Fragen der Wiederverwendung und Wartung ein.

CASE–Werkzeug: Programm zur Unterstützung der computergestützten Software–Entwicklung (CASE). Möglich ist üblicherweise die grafische Modellierung, Konsistenzprüfung und Im– bzw. Export–Möglichkeiten.

Common Object Request Broker Architecture (CORBA): Eine objektorientierte Middleware, die plattformübergreifende Protokolle und Dienste definiert. Sie wurde von der Object Management Group (OMG) entwickelt. CORBA ermöglicht das Erstellen verteilter Anwendungen in heterogenen Umgebungen (siehe: http://www.omg.org/gettingstarted/corbafaq.htm).

Deadlock (Verklemmung): Ein Zustand in einem Prozeßsystem, bei dem mehrere Prozesse in einer solchen Weise wechselseitig aufeinander Warten, daß letztendlich keiner von ihnen mehr fortschreiten kann (aus Eberl et al. 1995:289).

Debugger: Ein spezielles Computerprogramm mit dem die Ablaufverfolgung eines ausgeführten Programmes in einzelnen Schritten möglich ist.

Design Pattern: sind Beschreibungen von Kommunizierenden Objekten und Klassen, die generelle Designprobleme in einem bestimmten Kontext zu lösen (siehe Gamma, 1997).

Extensible Markup Language (XML): Eine Seitenauszeichnungssprache und Quasi-Standard zur Erstellung strukturierter Dokumente. Es wurde von einer Arbeitsgruppe entwickelt, die unter Schirmherrschaft des World Wide Web Consortium (W3C) steht. (siehe: http://www.w3.org/XML/)

Geheimnisprinzip (Information Hiding): Auf die Attributwerte eines Objekts kann nur über die Operationen des Objekts zugegriffen werden. Für andere Klassen und Objekte sind die Attribute und Attributwerte einer Klasse oder eines Objektes sowie die Realisierung der Operationen unsichtbar (aus Balzert, 2000:179).

Host: Bezeichnung von Rechnern mit jeweils eigenem Betriebssystem, die in einem Rechnernetz miteinander verbunden sind.

Hypertext Transfer Protocol (HTTP): Ein populäres Protokoll zur Datenübertragung. Die aktuelle Version 1.1 ist definiert in RFC 2616 (siehe: ftp://ftp.isi.edu/in-notes/rfc2616.txt).

Interface: In dieser Arbeit verstanden als ein Sprachelement, beispielsweise der Programmiersprache Java, das Namen, Rückgabewert und Argumente von Operationen, sowie Konstanten definiert. Sie unterstützen den Vererbungsmechanismus objektorientierter Sprachen, die ansonsten nur Einfachvererbung erlauben (siehe: http://java.sun.com/docs/books/tutorial/java/concepts/interface.html).

Interface Definiton Language (IDL): Wurde von der Object Management Group (OMG) entwickelt und wird von CORBA benutzt. Mittels der IDL können formale Spezifikation von Klassen und Objekte sowie sämtlicher Parameter und Datentypen erstellt werden. Diese Schnittstellenbeschreibungen sind dann umsetzbar in ein Objektmodell einer bestimmten Programmiersprache (siehe: http://www.omg.org/gettingstarted/omg_idl.htm).

Internet Inter-ORB Protocol (IIOP): Ein von der Object Management Group (OMG) standardisiertes Protokoll (Object Management
Group, 2004), um Interoperabilität für CORBA zu ermöglichen. CORBA-konforme Systeme müssen dieses Protokoll unterstützen.

Java: Objektorientierte Programmiersprache, die von der Firma *Sun Microsystems* entwickelt wurde (siehe: http://java.sun.com/).

Java Runtime Environment (JRE): Eine Software, die zur Ausführung von Java-Programmen notwendig ist.
(siehe: http://java.sun.com/j2se/desktopjava/jre/index.jsp)

Java Virtual Machine (JVM): die Software, die zur Ausführung von kompilierten Java-Progammen benötigt wird
(siehe: http://java.sun.com/docs/books/vmspec/).

Java Database Connectivity Interface (JDBC): Eine Universelle Schnittstelle zu verschiedenen relationalen Datenbanken (siehe: http://java.sun.com/products/jdbc/).

Logging Framework: Stellt Funktionen zur integrierten Protokollierung bereit. Eine Implementation ist Teil der Java Klassenbibliothek (`java.util.logging`), es gibt eine Anzahl frei verfügbarer Implementationen (Beispielsweise: Log4J - http://logging.apache.org/log4j/docs/).

Lesser General Public License (LGPL): Eine freie Lizenz der Free Software Foundation (FSF; siehe http://www.gnu.org/home.html). Die LGPL wurde hauptsächlich für Programmbibliotheken geschaffen und erlaubt, dass auch proprietäre Programmteile gegen eine, unter der LGPL stehende, Bibliothek gelinkt werden
(deutsche Übersetzung unter: http://www.gnu.de/lgpl-ger.html).

Middleware: Allgemeine Bezeichnung für eine (Software–)Schicht die in verteilten Systemen die Heterogenität der zugrunde liegenden Plattformen verbirgt (siehe van Steen und Tanenbaum, 2002:36).

Object Constraints Language (OCL): Spezifikationssprache der UML zur Formulierung von Bedingungen.

Object Query Language (OQL): Eine Anfragesprache für objektorientierte Datenbanksysteme. Spezifiziert durch die Object Data Management Group (ODMG; siehe: http://www.odmg.org/).

Ontologie: In dieser Arbeit wird der Begriff im Sinne der Künstlichen Intelligenz verstanden. Es bezeichnet ein formal definiertes System von Dingen und/oder Konzepten und Relationen zwischen diesen Dingen.

Remote Method Invocation (RMI): Ist ein Mechanismus der Programmiersprache Java und bezeichnet den Aufruf einer Methode eines (Java–) Objektes, dass sich dass sich in einer anderen *Java Virtual Machine* befinden kann (somit auch auf anderen Rechnern).Außerdem bezeichnet der Begriff das Kommunikationsprotokoll, das für diese Aufrufe verwendet wird (siehe: http://java.sun.com/products/jdk/rmi/).

Roboter: Autonome Maschinen, die selbstständig eine bestimmte Aufgabe erfüllen. Der Begriff wurde von dem Schriftsteller Karel Čapek (im Schauspiel *Rossum's Universal Robots*, 1920) eingeführt.

Template Engine: In dieser Arbeit werden hierunter Java–basierte Werkzeuge verstanden, die es erlauben Methoden von Objekten mittels einer eigenen Skriptsprache (Template Language) aus beliebigen Textdokumenten heraus zu erreichen und die Rückgabewerte in diese einzubinden. Dieser Mechanismus kann benutzt werden um dynamisch beliebige Text–Dateien zu generieren (Beispielimplementation: siehe die Velocity Template Engine unter: http://jakarta.apache.org/velocity/index.html).

Unified Modelling Language (UML): Notation für die Darstellung von Modellen im Software–Engineering. Aktuelle Informationen sind unter
http://www.uml.org/ zu finden.

Unified Process (UP): Eine Methode zur Softwareentwicklung. Er ist Anwendungsfall gesteuert, architekturzentriert, iterativ und inkrementell. Er besteht aus vier Phasen (*Etablierung*, *Entwurf*, *Konstruktion* und *Übergang*) in denen jeweils fünf Kernarbeitsschritte (Workflows) (*Anforderungen*, *Analyse*, *Design*, *Implementierung* und *Testen*) ausgeführt werden (Jacobson, Booch und Rumbaugh, 1999, Übersetzung der Begriffe aus Kahlbrandt, 2001 übernommen).

Verteilte Künstliche Intelligenz (VKI): Ziel der Verteilten Künstlichen Intelligenz ist es, Organisationen zu schaffen, die aus symbolverarbeitenden Systemen bestehen und Probleme durch logisches Schließen lösen können (aus Ferber, 2001:44).

Verteiltes System: Ein, aus einem Zusammenschluss unabhängiger Computer bestehendes, System (van Steen und Tanenbaum, 2002:2).

XML Schema: Eine formale Sprache, die es ermöglicht die Struktur wohlgeformter XML-Dokumente zu beschreiben (eine Einführung gibt W3C Recommendation, 2001a; Definition in W3C Recommendation, 2001b; 2001c).

Anhang F

Inhalt der beigelegten CD–ROM

Die im Rahmen dieser Arbeit erstellten Dateien sind auf der beigefügten CD–ROM enthalten.

Hier das Inhaltsverzeichnis des Datenträgers:

`Fallbeispiel`: Enthält die Modellierungen der Tutorial–Aufgabe E3 aus Kapitel 6

- E3.pd: Prometheus Design Tool Projekt–Datei
- MaSEProjekt-v2.0.maml: Projekt Datei für agentTool (2.0 beta)
- tropos.zargo: Tropos Modelle für das *Poseidon* UML CASE–Werkzeug (Gentleware[1] - Community Version frei erhältlich)

`Images`: Enthält die in der Diplomarbeit verwendeten Bild–Dateien.

`MarsMAS`: Enthält den Quellcode und Modellierung des Fallbeispieles aus Kapitel 7.

`Pdt2Adf`: Enthält den Quellcode des entwickelten Prototypen.

`Pdt2adfProject`: Enthält ein Eclipse–Projekt des entwickelten Prototypen.

`Quellen`: Enthält die verwendeten Quellen in elektronischer Form.

`DiplomarbeitDeckblatt.doc`: Deckblatt der Diplomarbeit, den Konventionen der HAW entsprechend.

`Zusammenfassung.doc`: Zusammenfassungsblatt zur Diplomarbeit, den Konventionen der HAW entsprechend.

`08.01.2004_Oberseminarvortrag.pdf`: Vortrag der im Oberseminar am Arbeitsbereich VSIS des Fachbereichs Informatik der Universität Hamburg gehalten wurde. Stellt Inhalt und Vorgehensweise der Diplomarbeit vor.

`10.12.2003_Seminarvortrag.pdf`: Vortrag der im Seminar "Agenten in verteilten Systemen." gehalten wurde. Gibt eine Einführung in agentenorientierte Methoden und stellt die Modellierung mittels der drei Methoden aus Kapitel 6 vor.

`diplom.pdf`: Die Diplomarbeit in elektronischer Form.

`Inhalt.txt`: Inhaltsverzeichnis des Datenträgers

Der Quellcode des Prototypen ist im CVS–System des Arbeitsbereiches Verteilte Systeme & Informationssysteme (VSIS) des Fachbereichs Informatik der Universität Hamburg verfügbar.

[1] http://www.gentleware.com

Anhang G

Bibliografie

Hier werden die in dieser Arbeit verwendeten Quellen aufgeführt. Sie werden alle im Text referenziert. Einige dieser Quellen werden dort nur der Vollständigkeit halber aufgeführt. So wurden die meisten, der in Tabelle 3.1 und in Abschnitt 5.1.1 referenzierten Methoden und Notationen nicht weiter betrachtet.

Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents^{TM} JACOB Manual*, 2003. www.agent-software.com

Apache Jakarta Project, *The Velocity User Guide*, 2003a. http://jakarta.apache.org/velocity/user-guide.html

Apache Jakarta Project, *Velocity Developer's Guide*, 2003b. http://jakarta.apache.org/velocity/developer-guide.html

Aridor Y. und Lange D. B. "Agent Design Patterns: Elements of Agent Application Design". In Proceedings of the second international conference on Autonomous agents, Minneapolis, Minnesota, United States, ACM Press, 1998.

Avison D.E. und Fitzgerald G. *Information Systems Development: Methodologies, Techniques and Tools*, 3 ed, McGraw Hill, 2003.

Balzert H. *Lehrbuch der Software–Technik - Software–Entwicklung*, 2. Auflage, ISBN: 3827403014 , Spektrum Verlag, 2000.

Bellifemine F., Caire G., Trucco T. und Rimasa G. *Jade Programmer's Guide*, 2000–2003.

Berard E. V. "A comparison of objekt-oriented methodologies". Technical report, Objekt Agency Inc, 1995.

Bergenti F. und Poggi A. "Exploiting UML in the Design of Multi-Agent Systems". In Proc. of the ECOOP - Workshop on Engineering Societies in the Agents World 2000 (ESAW'00), 2000.

Bernon C., Gleizes M. P., Peyruqueou S. und Picard G. "ADELFE, a Methodology

for Adaptive Multi-Agent Systems Engineering", in Third International Workshop 75 "Engineering Societies in the Agents World" (ESAW-2002), Madrid, 2002.

Booch G. *Object-Oriented Analysis and Design with Application*, Addison-Wesley Professional, 1994.

Bratman M. *Intention, plans and practical reason*, Harward University Press, 1987.

Braubach L. und Pokahr A. *Jadex Tutorial* - Release 0.9, 2003.
http://sourceforge.net/projects/jadex

Bräutigam L., *Software-Ergonomie: Beurteilung der Software-Ergonomie anhand des ISONORM-Fragebogens*, Gesellschaft Arbeit und Ergonomie - online e.V., 1999a. unter: http://www.sovt.de/download.html

Bräutigam L., *Beurteilung der Software-Ergonomie: Einsatz des ISONORM-Fragebogens in einem beteiligungsorientierten Verfahren*, Gesellschaft Arbeit und Ergonomie - online e.V., 1999b. unter: http://www.sovt.de/download.html

Bresciani P., Giorgini P., Giunchiglia F., Mylopoulos J., Perini A. *Troops: An agent-oriented software development methodology*, Technical Report DIT-02-0015, University of Trento, Department of Information and Communication Technology, 2002.

Brooks R. "Elephants Don't Play Chess". In "Journal of Robotics and Autonomous Systems", Volume 6:3-15, 1990.

Burmeister B. "Models and Methodology for Agent-Oriented Analysis and Design". In K. Fischer (ed.), Working notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, DFKI Document D-96-06 (1996).

Busetta P., Howden N., Rönnquist R. und Hodgson A. "Structuring BDI Agents in Functional Clusters". in N. R. Jennings and Y. Lesperance, editors, Intelligent Agents VI. Springer Verlag, Berlin, 1999.

Bush G., Cranefeld S. und Purvis M. "The Styx agent methodology". The Information Science Discussion Paper Series 2001/02, Department of Information Science, University of Otago, New Zealand, 2001.
http://divcom.otago.ac.nz/infosci

Bush G. W. "President Bush Announces New Vision for Space Exploration Program - Remarks by the President on U.S. Space Policy". Ansprache, NASA Headquarters, Washington, D.C., 14.01.2004.
http://www.whitehouse.gov/news/releases/2004/01/20040114-3.html

Caire G., Leal F., Chainho P., Evans R., Garijo F., Gomez J., Pavon J., Kearney P., Stark J. und Massonet P. "Agent oriented analysis using message/uml". In Agent-Oriented Software Engineering (AOSE), Montreal, 2001a.

Caire G., Leal F., Rodrigues J. und Inovação T. *Message: Methodology for Engineering Systems of Software Agents - recommendations on supporting tools*, Eu-

ropean Institute for Research and Strategic Studies in Telecommunications GmbH (EURESCOM) Technical Information, Project P907, EDIN 0224-0907, 2001b.

Cernuzzi L. und Rossi G. "On the evaluation of agent oriented modeling methods", In Proceedings of Agent Oriented Methodology Workshop, Seattle, 2002.

Collinot A., Drogoul, A. und Benhamou, P. "Agent Oriented Design of a Soccer Robot Team", In Proc. of the Second Intl. Conf. on Multi-Agent Systems, Kyoto, Japan, 1996.

Cortese E., Caire G. und Bochicchio R. "JADE Test Suite – USER Guide", 2004. Enthalten in: http://sharon.cselt.it/projects/jade/dl.php?file=JADE-testSuite-3.2snapshot.zip

Cossentino M. und Potts C. "A CASE tool supported methodology for the design of multi-agent systems" - The 2002 International Conference on Software Engineering Research and Practice (SERP'02), USA, 2002.

Dam K. H. *Evaluating and Comparing Agent-Oriented Software Engineering Methodologies*. School of Computer Science and Information Technology, RMIT University, Australia, Minor thesis submitted in partial fulfillment of the requirements for the degree of Master of Applied Science in Information Technology, 2003. http://yallara.cs.rmit.edu.au/~kdam/

Dam K. H. und Winikoff M. "Comparing Agent-Oriented Methodologies", to appear in the proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (at AAMAS03), Melbourne, 2003.

Debenham J. und Henderson-Sellers B. "Full lifecycle methodologies for agent-oriented systems – the extended OPEN process framework". In Proceedings of Agent-Oriented Information Systems (AOIS-2002) at CAiSE'02, Toronto, 2002.

DeLoach S. A. "Analysis and design using MaSE and agentTool". In Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS), 2001.

DeLoach S. A. "Using agentMom". Unpublished document. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. October 1999. unter: www.cis.ksu.edu/ sdeloach/ai/software/agentMom_2.0/Final_Report/CHAPTER_12.pdf

Dikenelli O. und Erdur R.C. "SABPO: A Standards Based and Pattern Oriented Multi-Agent Development Methodology". Third International Workshop on Engineering Societies in the Agents World, ESAW02, Universidad Rey Juan Carlos, September 16-17, Madrid, Spain, 2002.

DIN Deutsches Institut für Normung e.V., Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten - Teil 10: Grundsätze der Dialoggestaltung, DIN EN ISO 9241-10, Beuth Verlag, 1996.

Do T.T., Kolp M., Hang Hoang T. T. und Pirotte A. "A Framework for Design

Patterns for Tropos". In Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES 2003), Maunas, Brazil, October 2003.

Eberl H., Frank M., Jacobi I., Jaeger F., Kabitzsch K., Löffler H., Meinhardt J., Müller H., Pätzold W., Petersohn U., Pippig E., Schubert D., Strothotte T., Werner D., Würkert M. *Tachenbuch der Informatik*, Hrsg. Werner D., 2. Aufl., Fachbuchverlag Leipzig, ISBN 3-343-00892-3, 1995.

Far B. H. "Agent-SE: A Methodology for Agent Oriented Software Engineering". Enabling Society with Information Technology, Q. Jin et al. Edts., Springer, ISBN 4-431-70327-7, 2001.

Ferber J. *Les Système Multi–Agents. Vers une intelligence collective*. ©InterEditions, Paris, 1995. (Deutsche Übersetzung von Kirn S.: Multiagentensysteme – Eine Einführung in die Verteilte Künstliche Intelligenz, ©Addison–Wesley, 3-8273-1679-0, 2001).

Ferber J. und Gutknecht O. "A meta-model for the analysis and design of organizations in multi-agent systems". In Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98) Paris, France, 1998.

Foundation for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*, SC00001L, 2002.
http://www.fipa.org/specs/fipa00001/

Franklin S. und Graesser A. "Is it an Agent, or just a Program?". In Proceedings of the Third International Workshop on Agent Theories, Architektures, and Languages, Springer-Verlag, 1996.

Gamma E., Helm R., Johnson R. und Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1997.

Georgeff M. und Lansky A. "Reactive Reasoning and Planning: An Experiment With a Mobile Robot", in Proceedings of the 1987 National Conference on Artificial Intelligence (AAAI 87), Seattle, Washington, 1987.

Gervais M.-P. "ODAC: An Agent-Oriented Methodology Based on ODP". Accepted for publication in Journal of Autonomous Agents and Multi-Agent Systems in January 2002.

Giunchiglia F., Mylopoulos J. und Perini A. "The Tropos software development methodology: Processes, Models and Diagrams". In Third International Workshop on Agent-Oriented Software Engineering, 2001.

Glaser N. "The CoMoMAS Approach: From Conceptual Models to Executable Code". http://citeseer.nj.nec.com/, 1997.

Gomez-Sanz J.J., Fuentes R. "Agent Oriented Software Engineering with INGENIAS". Fourth Iberoamerican Workshop on Multi-Agent Systems (Iberagents 2002) — Agent Technology and Software Engineering, University of Málaga (Spain), 11-12 November 2002.

Henderson–Sellers B. und Gorton I. "Agent-based Software Development Methodologies", Whitepaper, Zusammenfassung eines Workshops an der OOPSLA 2002, 2003.
http://www.open.org.au/Conferences/oopsla2002/index.html

Howden N., Rönnquist R., Hodgson A. und Lucas A. "JACK Intelligent Agents^{TM}– Summary of an Agent Infrastructure". In 5th International Conference on Autonomous Agents, 2001.

Iglesias C.A., Garijo M. und González J.C. "A Survey of Agent-Oriented Methodologies". In Intelligent Agents V – Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

Iglesias C., Garijo M., González J. und Velasco J.R. "Analysis and Design of multiagent systems using MAS-CommonKADS". Intelligent Agents IV: Agent Theories, Architectures and Languages, Singh, M. P., Rao, A. and Wooldridge, M.J., eds., Lecture Notes in Computer Science 1365, 1998.

Jacobson I., Booch G., Rumbaugh J. *The Unified Software Development Process*. Object Technology Series. Addison Wesley Longman, Reading, MA, 1. Auflage, ISBN 0-201-57169-2, 1999.

Jennings N.R. und Wooldridge M. *Agent-Oriented Software Engineering*. Handbook of Agent Technology (ed. J. Bradshaw) AAAI/MIT Press, 2000.

Jennings N. R. "Building complex, distributed systems: the case for an agent-based approach". Comms. of the ACM, 44 (4), 2001.

Jennings, N.R., Faratin, P., Normam,T.J., O'Brien, P., Odgers, B. und Alty, J.L. "Implementing a Business Process Management System Using ADEPT: A Real-World Case Study", Intl. Journal of Applied AI 14(5), 2000.

Jennings N.R. "On Agent–Based Software Engineering". Artificial Intelligence, 117(2), 2000.

Jonker C.M., Klush M. und Treur J. "Design of Collaborative Information Agents". In M.Klush and L. Kerschberg (eds.), Cooperative Information Agents IV, Proc. Of CIA 2000. Springer, 2000.

Juan T., Pearce A. und Sterling L. "ROADMAP: Extending the Gaia Methodology for Complex Open Systems". In Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS2002), Bolognia, Italy, 2002:3.

Kahlbrandt B. *Software-Engineering mit der Unified Modeling Language*. Springer Verlag, 2. Aufl., ISBN 3–540–41600–5, 2001.

Kendall E. A., Krishna P. V. M., Pathak C. V. und Suresh C. B. "Patterns of

intelligent and mobile agents". In Proc. of the second international Conference on Autonomous agents, 1998.

Kendall E.A., Malkoun M.T. und Jiang C. "A Methodology for developing Agent Based Systems for Enterprise Integration". In D.Lukose and C.Zahng, editors, First Australian Workshop on Distributed Artificial Intelligence Springer-Verlag, 1996.

Kirkwood C. W. *Strategic Decision Making: Multiobjective Decision Analysis with Spreadsheets*. Belmont, California: Wadsworth Publishing, 1997.

Kitchenham B. *DESMET: A method for evaluating Software Engineering methods and tools*. Technical Report TR96-09, ISSN:1353-7776, 1996.

Kinny D., Georgeff M. und Rao A. "A methodology and modelling technique for systems of BDI agents". In W. Van de Velde and J.W. Perram, editors, Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038). Springer-Verlag: Berlin, 1996.

Knublauch H. "Extreme Programming of Multi-Agent Systems". Proc. of the First Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna, Italy (2002).

Langton C. *Artificial Life*, Reports / LOS ALAMOS NAT LAB NM, Report-Nr.: DE89 002240; LA UR 88 3397, 1988.

Liedekerke M. und Avouris N. "Debugging multi-agent systems". Information and Software Technology, 37(2), pg 102–112, 1995.

Lind J. "Agent-Oriented Software Engineering with MASSIVE". In Informatiktage 2001, Konradin Verlag, März, 2001.

Luck M., McBurney P. und Preist C. *Agent Technology: Enabling Next Generation Computing A Roadmap for Agent Based Computing*. AgentLink, ISBN 0854 327886, 2003. http://www.agentlink.org/roadmap/index.html

Mangina E. *Review of Software Products for Multi-Agent Systems*, Agentlink, 2002. Entwurfsversion zum Download unter: http://www.agentlink.org/resources/other-pubs.html

Messerschmidt L., "Take the fast track to text generation - Create text content with template engines and save time and frustration". Java World, July 2001. http://www.javaworld.com/javaworld/jw-07-2001/jw-0727-templates.html

Moulin B. und Brassard M. "A Scenario-Based Design Method and an Evironment for the Development of Multi-Agent Systems". In D.Lukose and C.Zahng, editors, First Australian Workshop on Distributed Artificial Intelligence Springer-Verlag, 1996.

Newell A. und Simon H. "Computer science as empirical inquiry: symbols and search". Communications of the ACM, 19(3), 1976.

Nielsen N. J.*Artificial Inelligence: A New Systhesis*. Morgan Kaufmann Publishers, Inc, 1998.

Nwana H., Ndumu D., Lee L. und Collis J. "ZEUS: A Toolkit for Building Distributed Multi-Agent Systems". Proceedings of the Third International Conference on Autonomous Agents (Agents'99), 1999.

Nwana S. H. "Agents: An Overview". The Knowledge Engineering Review Vol. 11, No 3, Cambridge University, 1996.

Object Management Group. *CORBA^{TM}/IIOP^{TM} Specification*, formal/04-03-12 (CORBA specification, v3.0.3), 2004.
http://www.omg.org/technology/documents/formal/corba_iiop.htm

Object Management Group. *UML Superstructure 2.0 Draft Adopted Specification*. OMG Docment ptc/03-08-02, 2003.

Object Management Group. *Mobile Agent Facility Specification*. v1.0, January, 2000.

Odell J. "Agent UML: What is It and Why Do I Care". Vortrag während der Net.ObjectDays 2003. http://www.jamesodell.com/What-is-UML.pdf

Odell J., Parunak H.V.D., Bauer B. "Extending UML for Agents". In G. Wagner, Y. Lesperance, and E. Yu, editors, Proceedings of the AgentOriented Information Systems Workshop (AOIS) at the 17th National Conference on Artificial Intelligence, 2000.

O'Malley S. A. und DeLoach S. A. "Determining When to Use an Agent-Oriented Software Engineering Paradigm". Proceedings of the Second International Workshop On Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada, 2001.

Omicini A. "SODA: Societies and infastructures in the analysis and design of agent-based systems". In P. Ciancarini and M. J. Wooldridge, editors, Proceedings of Agent-Oriented Software Engineering 2000 (AOSE 2000), 2000.

Padgham L. "Design of Multi Agent Systems". Tutorial at Net.ObjectDays, October 7-10, 2002, Erfurt, Germany, 2002.
http://www.netobjectdays.org/node02/de/Conf/publish/talks.html#Design_of_Multi_Agen.t

Padgham L. und Winikoff M. "Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents". In the proceedings of the workshop on Agent-oriented methodologies at OOPSLA 2002.

Pokahr A. und Braubach L. *Jadex User Guide* - Release 0.9, 2003.
http://sourceforge.net/projects/jadex

Pokahr a., Braubach L., Lamersdorf. W. "Jadex: Implementing a BDI-Infrastructure for JADE Agents". In: EXP - In Search of Innovation (Special Issue on JADE), Vol 3, Nr. 3, Telecom Italia Lab, Turin, Italy, September 2003.

Poutakidis D., Padgham L. and Winikoff M. "An exploration of bugs and debugging in multi-agent systems". in Proceedings of the second international joint conference on Autonomous agents and multiagent systems, ACM Press, ISBN: 1-58113-683-8, 2003.

Poutakidis D., Padgham L. und Winikoff M. "Debugging multi-agent systems using design artifacts: The case of interaction protocols". In Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02), 2002.

Prümper J., Anft M., Fragebogen ISONORM 9241/10, 1993.
http://www.sozialnetz.de/aweb/pq.asp?id=jsi oder zum Download unter:
http://www.sozialnetz.de/aweb/rd.asp?id=hmh

Rao A. und Georgeff M. "BDI-agents: from theory to practice". In Proceedings of the First Intl. Conerence on Multiagent Systems, San Francisco, 1995.

Rational Software White Paper *Rational Unified Process: Best Practices for Software Development Teams*, 2001a, unter: http://www.rational.com/products/whitepapers/100420.jsp

Rational Software White Paper *Business Modeling with the UML and Rational Suite AnalystStudio*, 2001b, unter: http://www.rational.com/uml/resources/whitepapers/index.jsp

Rational Software White Paper *Mapping Object to Data Modells with the UML*, 2000, unter: http://www.rational.com/uml/resources/whitepapers/index.jsp

Rational Software White Paper *Modeling Web Application Architectures with UML*, 1999, unter: http://www.rational.com/uml/resources/whitepapers/index.jsp

Rumbaugh J., Blaha M., Premerlani W., Eddy F. und Lorensen W. *Object–Oriented Modeling and Design*. Prentice–Hall, Englewood Cliffs, NJ, 1991.

Rumbaugh J., Jacobson, I. und Booch, G. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

Saba G. M., Santos, E. "The Multi-Agent Distributed Goal Satisfaction System". Proceedings of the International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA '2000), 2000.

Shehory O. und Sturm A. "Evaluation of modeling techniques for agent-based systems". In Jörg P. Müller, Elisabeth Andre, Sandip Sen, Claude Frasson, Hrsg., Proceedings of the Fifth International Conference on Autonomous Agents, ACM Press, 2001.

Studer R., Decker S., Fensel D. und Staab S., "Situation and Perspective of Knowledge Engineering". In: J. Cuena, et al. (eds.), Knowledge Engineering and Agent Technology. IOS Press, Amsterdam, 2000.

Sturm A. und Shehory O. "A Framework for Evaluating Agent-Oriented Methodologies". Workshop on Agent-Oriented Information System (AOIS), Melbourne, Australia, 2003.

Sturm A., Dori D. und Shehory O. "Single-Model Method for Specifying Multi-Agent Systems". In Proceedings of the Second International Conference on Autonomous Agents and Multi– Agent Systems (AAMAS 2003), Melbourne, Australia, 2003.

Sudeikat J. Braubach L., Pokahr A. und Lamersdorf W. "Evaluation of Agent–Oriented Software Methodologies – Examination of the Gap between Modeling and Platform". Angenommen am Fifth International Workshop on AGENT-ORIENTED SOFTWARE ENGINEERING (AOSE-2004), 2004.

Tveit A. "A survey of Agent-Oriented Software Engineering". In: NTNU Computer Science Graduate Student Conference, Norwegian University of Science and Technology, Trondheim, Norway, 2001.

van Lamsweerde A. "From System Goals to Software Architecture". In Formal Methods for Software Architectures, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003.
//ftp.info.ucl.ac.be/pub/publi/2003/avl-ReqToArch-avl.pdf

van Steen M., Tanenbaum A.S. *Distributed Systems - Principles and Paradigms*, Prentice Hall, 2002.

W3C Recommendation *XML Schema Part 0: Primer*, Fallside D. C. (editor), 2001a.
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

W3C Recommendation *XML Schema Part 1: Structures*, Thompson H. S., Beech D., Maloney M., Mendelsohn N. (editors), 2001b.
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

W3C Recommendation *XML Schema Part 2: Datatypes*, Biron P. V., Malhotra A. (editors), 2001c.
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

Wagner G. "The Agent-Object-Relationship Meta-Model: Towards a Unified View of State and Behavior". Information Systems 28:5, 2003.

Wood B., Pethia R., Gold L.R. und Firth R. *A guide to the assessment of software development methods*, Technical Report 88-TR-8, Software Engineering Institute, Carnegie- Mellon University, Pittsburgh, PA, 1988.

Wood M. F. und DeLoach S. A. "An Overview of the Multiagent Systems Engineering Methodology". The First International Workshop on Agent-Oriented Software

Engineering (AOSE-2000), 2000.

Wooldridge M. und Ciancarini P. "Agent–Oriented Software Engineering: The State of the Art". in Handbook of Software Engineering and Knowledge Engineering: World Scientific Publishing Co., 2001.

Wooldridge M. J., Jennings N. R. und Kinny D. "The Gaia methodology for agent-oriented analysis and design". Autonomous Agents and Multi-Agent Systems, 2000.

Wooldridge M. J. und Jennings N. R. "Software Engineering with Agents: Pitfalls and Pratfalls". IEEE Internet Computing, 3(3):20-27, May/June 1999.

Wooldridge M. J., Jenning N. R. und Kinny D. "A methodology for agent-oriented analysis and design". In Proc. of the third international conference on Autonomous agents, 1999.

Wooldridge M. J. und Jennings N. R. "Pitfalls of agent-oriented development". In Proc. of the second international conference on Autonomous agents, 1998.

Wooldridge M. J. und Jennings N. R. "Intelligent Agents: Theory and Practice". The Knowledge Engineering Review, 2(10), 1995.

Yim H., Cho K., Jongwoo K. und Park S. "Architecture-Centric Object-Oriented Design Method for Multi-Agent Systems". In Proc. of the Fourth International Conference on MultiAgent Systems (ICMAS-2000), 2000.

Yu E. und Cysneiros L. M. "Agent-Oriented Methodologies - Towards A Challenge Exemplar". Electronic Edition (CEUR Workshop Proceedings), 2002.
unter: www.cs.toronto.edu/pub/eric/AOIS02.pdf

Yu, E. "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering". In Proc. of the 3rd IEEE Int. Symp. on Requirements Engineering, 1997.

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(4) bzw. §35(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 26 Mai 2004

_____

(Jan Sudeikat)